The Coq proof assistant user's guide : version 5.8

Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Catherine Parent,
Christine Paulin-Mohring, Benjamin Werner, Chetan Murthy

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The Coq Proof Assistant User's Guide

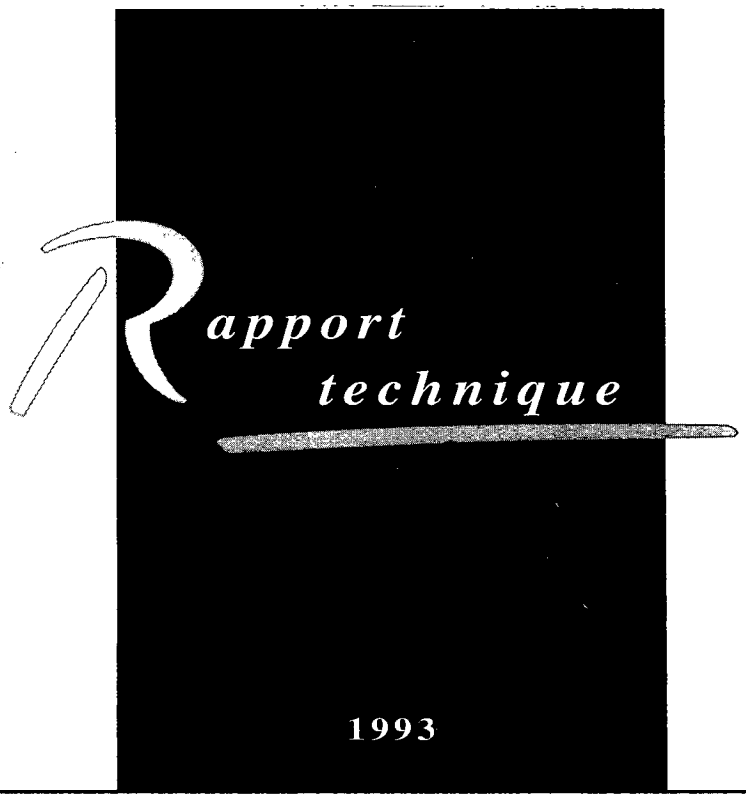
Version 5. 8

Gilles DOWEK - Amy FELTY - Hugo HERBELIN
Gérard HUET - Chet MURTHY - Catherine PARENT
Christine PAULIN-MOHRING - Benjamin WERNER

N° 154
Mai 1993

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel



The Coq Proof Assistant
User's Guide

Version 5.8 *

Gilles DOWEK – Amy FELTY – Hugo HERBELIN

Gérard HUET – Chet MURTHY – Catherine PARENT

Christine PAULIN-MOHRING – Benjamin WERNER

Projet Formel

INRIA-Rocquencourt† — CNRS - ENS Lyon ‡

*This research was partly supported by ESPRIT Basic Research Action "Types" 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.

1st Printing February 1st, 1993

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. Exploiting this Curry-Howard isomorphism, 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 T. Coquand and C. 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.

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. A less ambitious approach to proof developement is computer-aided proof-checking. The most notable 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.

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. 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.

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 1989. Then, the system was 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 was streamlined, and the vernacular extended for tactics execution. A package to compile programs extracted from proofs to actual computer programs in CAML or some other functional language was designed and implemented by B. Werner. A new user-interface, relying on a CAML-X interface by D. de Rauglaudre, was

designed and implemented by A. Felty. It allowed operation of the theorem-prover through the manipulation of windows, menus, mouse-sensitive buttons, and other widgets. This system (Version 5.6) was realeased in 1991. This new version 5.8 whose development has been coordinated by C. Murthy is written in CAML-LIGHT (Version 0.5), a new implementation of CAML. It provides new tools designed by C. Parent to prove properties of ML programs (this methodology is dual to program extraction) and a new user-interaction loop.

The first part of this manual is a user's guide to the theorem-proving facilities. The second part explains how to extract ML programs from Coq proofs and prove properties of ML programs. At last an appendix contains the libraries initially loaded in the system.

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 ftp.inria.fr*
Connected to ftp.inria.fr.
220 ftp FTP server ready.
Name (ftp.inria.fr:huet):†
Password: myself@mymachine.myinstitution.mycountry
331 Guest login ok, access restriction apply
ftp> cd INRIA/coq/V5.8
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`, `RELEASED`, `THEORIES`, `LIB`, `DEMO`, and a number of files: `README`, `coql`. `DOC` contains the dvi format of this manual, `SRC` contains the CAML-Light source files, `THEORIES` contains a selection of example vernacular files, comprising the mathematics and programs developed in the system. `RELEASED` will contain entries for every machine architecture you want to run Coq on. For instance, if you are on a sun4 machine, the installation of the system as explained below will create an entry `sun4/coq.out`.

Getting the system to run

In order to run the system, you need CAML-Light version 0.5. CAML-Light may also be obtained by FTP. Do, similarly as above: `ftp> cd /lang/caml-light`

```
250 CWD command successful.
ftp> get README
```

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

†here you type in **anonymous**

```
ftp> get cl5unix.tar.Z
ftp> get rt5.tar.Z
```

The file `cl5unix.tar.Z` is the CAML-Light 0.5 UNIX distribution, and the file `rt5.tar.Z` is the X-windows runtime environment, used for the X interface to Coq.

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-Light, and the X runtime library installed, you have to install Coq. In directory `coq/LIB/libc`, do `make`. In directory `coq/LIB/stream-pp`, do `make`. Finally, in directory `coq/SRC`, do `make` to create the executable image. This will create three files: `coq`, `xcoq`, and `state.coq`, which are, respectively, the character-based I/O executable, the X-windows-based I/O executable, and the saved internal state after loading the standard prelude of mathematical definitions. Finally, you will need to move these binaries to the `RELEASED` repository for your architecture. Create the proper repository, `mkdir RELEASED/'arch'`, and then, from within SRC, do: `make install`. If your system does not have an "arch" command, you will need to set the Makefile variable `ARCH` by hand, to your architecture. This completes the installation.

Once Coq is installed, you may run it by calling the command `coql` from the directory `coq`. You should get the banner:

```
Welcome to Coq V5.8 - Fri Jan 15 15:24:31 MET 1993
Coq <
```

The options of this command are :

- `-x` (boots version with X interface)
- `-bw` (only valid with `-x`, by default the color mode is used on color screens and the black and white mode on black and white screens, this option is needed to use the black and white mode on color screens)
- `-gray` (only valid with `-x`, if a gray scale screen is used)
- `-is file` (read initial state from `file`)
- `-I dir` (add `dir` to loadpath)
- `-q` (no autoload of `~/.coqrc`)

Navigating in the system

Initially, you are in the Coq toplevel. From this toplevel, you may exit permanently in a graceful manner by typing `Quit.`, or by typing CTRL-D. The keyword `Drop.` drops you into a fake CAML toplevel, which is used by system implementors for debugging. If you drop into it (the prompt is `comm #`), you can get back to the Coq toplevel by typing `go();;`.

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 Coq session with the shell command `fg`.

Interaction with the prover is usually done from the Vernacular toplevel, which is the default toplevel, when you invoke Coq. 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.

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 1.4.

It is possible to drive the theorem prover from a visual interface with the X window system. If you have a color screen, button of the visual interface will be colored.

You enter the interface from the `Coq` loop with the `Interface.` command. 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 1.6.

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 2.4 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.

Users Contributions

We are encouraging users to contribute their development examples so that some sharing of effort will be possible at the level of theory packages. These exemples will be included progressively in the released examples.

The current structure of the theories is as follows. The main installation directory of Coq V5.8 corresponds to an environment variable `$COQTOP`. The directory of theory packages corresponds to an environment variable `$COQTH`, by default equal to `$COQTOP/THEORIES`. Various packages are placed there, as subdirectories `SETS`, `LISTS`, `ARITH`, `RELATIONS`, `STREAMS`, etc. The directory `PROGRAMS` contains the examples of program proofs developed by the implementation team, in Lyon and in Rocquencourt. The directory `SYSTEM` contains the vernacular files initially loaded in the system, i.e. `Prelude`, `Specif`, `Peano` (natural numbers) and `Wf` (nœtherian induction). as well as certain other files of frequent use. The directory `ARITH` contains the beginning of integer arithmetic. Initially, the system's `LoadPath` is initialized to `[$COQTH/SYSTEM; $COQTH/ARITH]`.

The directory `CONTRIB` is intended for user-contributed examples, partitioned by sites. Every site will be responsible for his own sub-directory, but we propose a common format for consistency. Let us take the example of a site `Nancy`, who contributes packages for unification and for constraints, sharing a library of term structures. Typically, the directory `$COQTH/SYSTEM/CONTRIB/Nancy` will contain 3 subdirectories, say `Unify`, `Constraints`, and `Terms`. We propose that each of these directories contains a file `paths.v`, containing the proper LoadPath adjustements. Typically, the file `Unify/autoload.v` may contain:

```
AddPath "$COQTH/SYSTEM/LISTS".
AddPath "$COQTH/CONTRIB/Nancy/Terms".
AddPath "$COQTH/CONTRIB/Nancy/Unify".
```

There should also be a **README** file containing credits, disclaimers, confidentiality requests, origin of the example, explanations on the axiomatisation style, need of special features, etc. Finally, a file `load.v` ought to contain the sequence of **Require** commands necessary to load the package.

Each vernacular file ought to start with a comment header, indicating:

- A brief description of the nature of the theory
- Credits: authors, institutions, literature references used, etc.
- The version of the system in which the example was developed, e.g. `Coq V5.8`
- The date at which this theory was contributed

Each file should start with the minimum **Require** commands necessary to run this particular file, in the proper order. There is no need to do the transitive closure of this operation, of course, but redundancy is not problematic, since each module will be loaded only once. The file should end with the proper **Provide** command. It is good practice to end each package with a **postlude.v** file, which closes all opened sections and other global resettings.

We warn the contributing users to check carefully that their examples are executable with a virgin system. In particular, initialisation commands which the user has put in his personal `.coqrc` file ought to be copied in the proper `autoload.v` files.

If the development is documented in a research article or report, it may be useful to include in the directory this document, in the form of a **dvi** or **ps** file, if it is not too big. The next paragraph gives practical directions on how to contribute a Coq library.

The Coq librarian is **Gerard.Huet@inria.fr**. A first contact should be done with him by email to discuss the opportunity of contributing a given library. When an agreement has been reached, here are the steps to follow:

- Proceed with formatting the library according to the above directions
- Check carefully that the library loads without error in the current distribution version of Coq, with empty `.coqrc`.
- Remove from the corresponding directories all spurious items, such as temporaries, old versions and editor checkpoints files, including ones whose names start with a dot.
- make a tar archive, compress it, and uuencode it
- send it by email, preceded by a warning message "the next mail contains bla bla" with potential specific installation directions

All contributing authors will be regularly informed of new contributions, by concise messages sent on a coq-users-club distribution list. Please avoid mail pollution by global replies to such messages.

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 0.
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 `0`.

A shorter syntax for these declarations is:

```
Parameter nat:Set.
Parameter 0: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 0)`.
- 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.) Remark 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 -> nat -> 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`.

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

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`.

Remark 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 identifier `(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 the scope 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.
   Variable 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.
   Variable 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.
   Variable 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.

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.

Remark. If you try to execute all the examples given in this manual in a sequential manner, the above definition will fail with a message:

```
Error Clash with global variable nat
```

This is because `nat` was initially defined as an abstract type, represented as a global `Set` variable. Coq forbids to hide such global variables, and thus you must execute the above type definition in a fresh environment, obtainable by typing in:

```
Reset Initial.
```

Note that Coq allows redefinition of defined constants. Thus in the initial state of the system the type `nat` is defined from the standard prelude, but we may hide it by such a redefinition. A further remark is that in Coq inductive types are non-generative. The new type definition being isomorphic to the Prelude's one, these two types are internally equivalent, and thus arithmetic operators from the prelude such as `plus` may be applied to terms built with the "new" constructors.

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:

```
Definition 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` respectively.

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 concrete syntax 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 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 having expressed objects we can express 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.

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.

1.2.2 Connectives and Quantifiers

Propositions are built from atomic propositions with connectives and quantifiers.

- If P and Q are two propositions then P -> Q is the proposition "P implies Q".
- 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 proposition where a free variable x of type T may occur then (x:T)P is the proposition "for all x in T, P".
- If P is a proposition where a free variable x of type T may occur then <T> Ex ([x:T]P) is the proposition "there exists an x in T such that P".

For instance (Lower_eq 0 0) -> (Lower_eq 0 0) and (x:nat)(Lower_eq 0 x) are propositions. The latter is written in usual mathematical notation $\forall x \in \mathcal{N}\ 0 \leq x$.

In Coq it is possible to quantify over any data type. For instance one can quantify over a variable of type nat -> nat: (f:nat -> nat)(Lower_eq 0 (f 0)), one can also 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.

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. This overloading is in fact the sign of a deep isomorphism between types and propositions: the Curry-Howard isomorphism.

1.3 Assuming Axioms and Proving Theorems

1.3.1 Assuming Axioms

Axioms can be assumed using 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).
```

1.3.2 Proving Theorems

Theorems and their proofs are expressed with the instructions **Theorem**, **Statement**, **Goal** and **Save**.

For instance let us express and prove the theorem B.

```
Theorem b.
Statement B.
Goal.
  Apply u.
  Apply v.
Save.
```

With the lines `Theorem b.` and `Statement B.` we declare the name (b) of the theorem and its statement (B). Then we need to prove this theorem. We begin the proof with the instruction **Goal**. In this proof search phase we want to get as much help from the system as possible. In the spirit of the system LCF we give to the system some commands (tactics) which transform the initial problem into simpler problems (subgoals). In the previous example, the command **Apply u** transforms the goal B into the goal A using the axiom u of statement `A -> B`. The system displays the current status of the proof search as follows:

```
1 subgoal A
```

Then the command **Apply v.** proves this subgoal using the axiom **v** of statement **A**. The current status of the proof search is displayed

```
Goal proved!
```

Then we end the proof with the instruction **Save.**

The part of the system which provides assistance during proof search (by transforming the current goal into derivated goals and displaying the status of the search) is called the *Tactic Theorem Prover*. This Theorem Prover is described in the section 1.4.

1.3.3 Local Variables and Local Definitions in Axioms and Theorems

Variables can be declared and object can be defined with a scope restricted to an axiom or a theorem. These local declaration are inserted between the **Axiom** and the **Assumes** lines or between the **Theorem** and the **Statement** lines. Outside the scope of these declarations and definitions the local variables are universally quantified in the statement of the axiom or the theorem and the occurrences of the defined symbols are replaced by the term they abbreviate. For instance

```
Theorem th1.
   Variable A:Prop.
Statement A -> A.
Goal.
   Intro H.
   Apply H.
Save.
```

defines the theorem `th1 : (A:Prop)(A -> A).`

The keyword **Lemma** is synonymous with **Theorem**, and permits to reserve the terminology "Theorem" to important properties one wants to emphasize.

1.3.4 Hypotheses and Remarks

In the process of proving a theorem we may want to use intermediate lemmas, which we do not want to make available outside of the scope of the theorem. Such local lemmas can be defined, using the keyword **Remark** instead of **Theorem**.

In the same way local axioms (hypotheses) can be assumed using the keyword **Hypothesis** instead of **Axiom**. When an axiom or a theorem of statement `P` is declared with an hypothesis `A`, when this hypothesis is out of scope the statement of this axiom or theorem is transformed into `A -> P`. For instance

```
Theorem th2.
   Variable A:Prop.
   Hypothesis h.
   Assumes A.
Statement A.
Goal.
   Apply h.
Save.
```

defines the theorem `th2 : (A:Prop)(A -> A).`

As usual local variables, definitions, lemmas and theorems can be shared using the paragraph mechanism.

Summary of the Instructions

	Declarations	Definitions	Axioms	Theorems
Global	`Parameter`	`Definition`	`Axiom`	`Theorem/Lemma`
	`Inhabits`	`Body`	`Assumes`	`Statement`
				`Goal ... Save`
Local	`Variable`	`Local`	`Hypothesis`	`Remark`
	`Inhabits`	`Body`	`Assumes`	`Statement`
				`Goal ... Save`

1.4 The Tactic Theorem Prover

The Tactic Theorem Prover is the part of the system which permit to develop a proof by transforming the current goal into derivated goals. When a tactic is applied 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.

In a given state of the search, several subgoals may remain to be proved. To each of them is associated a local context. The essential steps of a proof search are the application of lemmas, theorems or axioms, reasoning by induction, and reduction of a constant to its definition. Moreover, some automatic proof search is provided. Thus the tactics can be organized in several categories.

- 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 are used when a goal is exactly an already proved theorem, an axiom or an hypothesis.
- 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.
- Compound tactics.

  These high-level tactics try automatic proof-search by combining more elementary tactics.

- Tacticals.

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

Let us start a proof development

```
Theorem S.
Statement (A,B,C:Prop)(A -> B -> C) -> (A -> B) -> A -> C.
Goal.
```

The command `Show` displays the current status of the proof.

```
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 by an hypothesis: Assumption

The first one, to which the tactics apply by default, is directly provable by the hypothesis `H1`. 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.

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

Backtracking: Undo

Backtracking is done by `Undo`.

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

```
    C : Prop
    B : Prop
    A : Prop
subgoal 2 is:
  B
```

Now, we have two subgoals again to solve. Let us continue the proof.

```
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 failure is to use **Try** which catches the failure of a tactic if it occurs. We invoke now the 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!

Coq < Save.
```

Elimination of inductive constants: Elim

The **Elim** tactic is used to prove a goal by induction over structure of a term of inductive type.

For instance, let us consider an inductive type

```
Inductive Set Color = blue:Color | red:Color.
```

and let **x** be a variable of type **Color**. We want to prove by cases the goal:
`(<Color>x = blue) \/ (<Color>x = red).`

The tactic **Elim** permits to prove this statement by cases.

```
Coq < Elim x.
2 subgoals
  (<Color>blue=blue)\/(<Color>blue=red)
subgoal 2 is:
  (<Color>red=blue)\/(<Color>red=red)
```

The goal has been transformed in two subgoals, the first corresponding to the case `blue` and the second to the case `red`.

When some of the constructors have arguments (as for instance the type of natural numbers which has two constructors `O:nat` and `S:nat -> nat`) the subgoals are more complicated. The elimination of `n` in the goal `(P n)` leads to two subgoals: the case where `n` has the form `O` leads to the subgoal `(P O)`, but the case where `n` has the form `(S m)` leads to the goal `(P (S m))` *under the hypothesis* `(P m)`, i.e. to the goal `(m:nat)(P m)->(P (S m))`. So we need to prove that the property `P` is verified by `O` and is hereditary. Induction on natural numbers appears here as a particular case of elimination.

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!
```

1.4.2 Focusing on a subgoal

The command `Focus` permits to have only one goal printed at a time, in Verbose mode:

```
Focus 2.
```

will print only the second goal. The command `Unfocus` disables focusing.

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 state as the one we got by doing `Intros`, then `Elim H`.

Dealing with connectors and quantifiers

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 Coq gives 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 is defined
```

Dealing with equality

Leibniz' equality is a constant which is defined by means of the 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 occurrences of u in t. The choice of the occurrences to be rewritten is described in the paragraph *Abstraction of Terms* below.

```
Coq < Theorem S_equal.
Coq < Statement (n,p:nat)(<nat>p=n)->(<nat>(S p)=(S n)).
Coq < Goal.
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 is defined
```

Dealing with symmetry of equality

Consider the following closely related problem:

```
Coq < Theorem sym_equal_elim.
Coq < Statement (A:Set)(x,y:A)(<A>x=y)->(P:A->Prop)(P y)->(P x).
Coq < Goal.
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 the **Trivial** tactic, i.e. by an hypothesis, an axiom or a theorem which generates no 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 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 = O : nat | S : nat -> nat.

Coq < Theorem O_or_Successor.
Coq < Statement (n:nat)(<nat>n=O)\/(<nat>Ex([n':nat](<nat>n=(S n')))).

Coq < Goal.
```

```
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.
0_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 < Theorem plus_0_commut.
Coq < Statement (n:nat)(<nat>(plus n 0)=(plus 0 n)).
Coq < Goal.

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 0) 0))=(S (plus m 0))
  ============================
    H : <nat>(plus m 0)=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 0)`, 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 0))` 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 0))` is the goal G in which only the second occurrence of m is substituted by `(plus m 0)`.

```
Coq < Undo.
1 subgoal
  <nat>(S (plus m 0))=(S m)
  ============================
    H : <nat>(plus m 0)=m
    m : nat
    n : nat

Coq < Pattern 2 m.
1 subgoal
  ([n:nat](<nat>(S (plus m 0))=(S n)) m)
  ============================
    H : <nat>(plus m 0)=m
    m : nat
    n : nat

Coq < Elim H.
1 subgoal
  <nat>(S (plus m 0))=(S (plus m 0))
  ============================
    H : <nat>(plus m 0)=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 is defined
```

1.4.3 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 is to give directly a proof-term of the goal. If the goal is an already proved theorem, an already proved remark, an axiom or an hypothesis the name of this item is a proof-term for the goal. More generally, the syntax for proof-terms is exaplained in the section 1.8.

- `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.

  Its use is as follows. When the current goal is reduced to a meta-variable, we may solve it by giving an explicit term `M` as solution, by the command:
  `Instantiate M.`

  This will replace the corresponding meta-variable by `M` in every other goal. 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 let us define a symbol `not_not_not`

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

  and consider the 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
```

The symbol `not_not_not` has 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 synthetized 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` above).

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.

Consider for instance the 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 we can chose to instanciate the second subgoal first.

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

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

- `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:

  ```
  (<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` $name_1$`:=`$term_1 \ldots name_n$`:=`$term_n$

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

  For example consider the 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` *term*

  There are two cases. When *term* is an identifier *name*, corersponding to a variable of the context (local or global) on which depends the current subgoal `G`, it replaces it by the same subgoal but quantified by the variable of name *name*.

  For example consider the 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
```

When *term* is not an identifier, and is a term of type T, the generated subgoal is T->G.

```
Coq < Generalize x1 x2 ... xn.
```

is equivalent to `Generalize xn; ..; Generalize x2; Generalize x1`. For instance, it generates the subgoal `(x1:T1)..(xn:Tn)G` if the `xi`'s are variables.

- `Specialize` *term*

  Here is a common situation. Assume you enter the following sequence of commands:

```
Parameter A:Prop.
Parameter R:nat->nat->Prop.
Parameters a,b:nat.
Goal ((x,y:nat)(R x y)->(R y x))->(R a b)->A.
Intros H H0.
```

  We are now in the situation:

```
Coq < 1 subgoal
  A
  ============================
    H0 : (R a b)
    H : (x:nat)(y:nat)(R x y)->(R y x)
```

  We may now change the current goal `A` into `(R b a) -> A` using any of the three following invocations of tactic `Specialize`:

```
Specialize (H a b H0).
Specialize H with x:=a y:=b 1:=H0.
Specialize H with 1:=H0.
```

  In the first case, `Specialize` acts like `Generalize`. In the second case, arguments are given by name, like in the `Apply` command, with numbers naming the anonymous assumptions of `H`, similarly to the `Induction` notation (remember that implication and quantification are both products, respectively non-dependent and dependent). Finally, in the third case, the arguments `x` and `y` are synthesised from the type of `H0`.

  It is also possible to specify with an optional argument the number of products which are instanciated in the specialized hypothesis. For instance, assume we are in the following situation:

```
1 subgoal
  <nat>x=z
  ============================
    H1 : <nat>y=z
    H0 : <nat>x=y
```

```
  z : nat
  y : nat
  x : nat
  H : (x:nat)(y:nat)(z:nat)(<nat>x=y)->(<nat>y=z)->(<nat>x=z)
```

If we want to instanciate H only in x and y, we may invoque:

```
Coq < Specialize 2 H with 1:=H0.
1 subgoal
  ((z:nat)(<nat>x=y)->(<nat>y=z)->(<nat>x=z))->(<nat>x=z)
  ============================
    H1 : <nat>y=z
    H0 : <nat>x=y
    z : nat
    y : nat
    x : nat
    H : (x:nat)(y:nat)(z:nat)(<nat>x=y)->(<nat>y=z)->(<nat>x=z)
```

Tactics for connectors, quantifiers and equality

- `Split`

  This tactic transforms the goal `A /\ B` into the two goals `A` and `B`.

- `Elim H` when `H` is an axiom, an hypothesis or an already proved theorem of the forms `A /\ B` transforms the goal `C` into the subgoals `A -> B -> C`. Combined with `Intro`, this tactic permit to use an know statement of the form `A /\ B` to deduce the hypotheses `A` and `B`.

- `Left`

  This tactic transforms the goal `A \/ B` into `A`.

- `Right`

  This tactic transforms the goal `A \/ B` into `B`.

- `Elim H` when `H` is an axiom, an hypothesis or an already proved theorem of the forms `A \/ B` transforms the goal `C` into the two subgoals `A -> C` and `B -> C`. In other words knowing that `A \/ B` is true, this tactic permits to prove a goal `C` by considering the two cases in which `A` and `B` are true.

- `Exists` *term*

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

- `Elim H` when `H` is an axiom, an hypothesis or an already proved theorem of the forms `<T> Ex [x:T]P` transforms the goal `C` into the two subgoals `(x:T)P -> C`. Combined with the tactic `Intro`, this tactic permits to give a name to an object whose existence has been proved.

- `Reflexivity`

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

- `Symmetry`

  This tactic reduces a goal `<T>x=y` to the goal `<T>y=x`.

- `Transitivity` *term*

  This tactic reduces a goal `<T>x=y` to the goals `<T>x=`*term* and `<T>`*term*`=y`.

Remark. These tactics also apply for variants of connectors, quantifiers and equality which will be presented in the cahapter 2 (as `*`, `+`, etc.).

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)` in order to do first order matching. In this case, it works as if `Elim` does `Pattern t1 ... tn` first.

- `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 nth 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 whose conclusion is an equality `<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.

  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`).

`Absurd` and `Contradiction`

- The tactic `Absurd` permits to prove a statement by contradiction.

  `Absurd` *statement*

  The current 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` *statement*

- The tactic `Contradiction` attempts to find in the current hypotheses (after all `Intros`) one which is equivalent to `False`. It permits to prune irrelevant cases. A typical use is:

```
Induction H; Simpl; Try Contradiction.
```

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 file `Peano.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"). Consider the goal
  `(~(~(~(refl nat gt))))-> ~(refl nat gt).`

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

- `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.

  For example consider the 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 consider a variable x of type nat and a 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*
- **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 !

Another tactic applying on hypotheses

- **Clear** $name_1 \ldots name_n$ This clears successively the hypotheses $name_1 \ldots name_n$ in the local context of the current subgoal.

  A cleared hypothesis cannot be used any more in the rest of the proof, so this tactic should not be used to simplify the display, but only when an hypothesis is really irrelevant.

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: 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` $name_i$, ..., (or `Exact` $name_i$ when the type of $name_i$ is not a product) to the hint list. A priority is assigned to each hint: 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` $name_1 \ldots name_n$

  This command adds the theorems or axioms $name_1 \ldots name_n$ to the hint list, with the indication to use them 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.

  `Immediate` declarations have priority 1.

- `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 $name_1 \ldots name_n$ (when the subgoal has the form ($name_i$ ...)).

  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 tactic and priority are 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 sumbool -> Apply right,1 Apply left,1
For le -> Apply le_n,0 Apply le_S,1
For lt -> Unfold * lt,4
For prod -> Apply pair,2
For and -> Apply conj,2
For sumor -> Apply inright,1 Apply inleft,1
For or -> Apply or_intror,1 Apply or_introl,1
For eq -> Apply plus_n_Sm,0 Apply plus_n_O,0 Apply refl_equal,0
          Apply eq_add_S ; Trivial,1 Apply eq_S,1 Apply sym_equal ; Trivial,1
For True -> Exact I,0
For not -> Apply n_Sn,0 Apply O_S,0 Apply not_eq_S,1
           Apply sym_not_equal ; Trivial,1
For ge -> Unfold * ge,4
For sum -> Apply inr,1 Apply inl,1
For gt -> Unfold * gt,4
```

- `Program`: see section 2.3.

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.

- $tactic$; [$tactic_1$ | ... | $tactic_n$]

  This construction applies $tactic_1$ to the first subgoal generated by $tactic$, ..., $tactic_n$ to the n^{th} subgoal generated by $tactic$. It requires that the number of subgoals generated by the application of $tactic$ is exactly the number of tactics in brackets.

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

  This compound 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.

- `Idtac` is the identity tactic.

1.4.4 Handling the goal environment

- *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.

- The command `Clear` permits to erase useless hypotheses. For instance

  ```
  Clear H1 y H3.
  ```

  erases the hypotheses `H1`, `y` and `H3`.

- The command `Focus` permits to have only one goal printed at a time, in Verbose mode:

  ```
  Focus 2.
  ```

  will print only the second goal. This is specially useful for big proofs, when there are numerous subgoals, each with a lengthy list of local hypotheses. In such cases the X interface usually becomes too slow, and the standard interface must be used, focusing on one subgoal at a time, and pruning the local hypotheses with the above `Clear` command. Typically, one adjusts an inductive statement on the inductive case with `Focus 2`, and on the base case with `Focus 1`.

  Finally, the following command disables focusing.

  ```
  Unfocus.
  ```

  Caution: `Focus 2` does not dispense from the initial `2:` for attacking the second goal. The user must also be conscious of the fact that this absolute naming of goals does not reflect the structure of goals that is, when subgoal 2 and all its generated subgoals will be solved, the original third subgoal will then be the second one.

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 files containing tactics.

- `Begin Silent`

  This command turns off the displaying.

- `End Silent`

  This command turns the normal displaying mode on.

1.5 Miscellaneous commands of the Coq system

1.5.1 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*).

  This command can also be used as `Check` *term*. In this case it takes a term and displays its type or a proof-term and displays the proved statement (see 1.8).

- `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.

1.5.2 Resetting 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 Empty`

  This resets the system to the state where only logical connectives, equality, booleans, natural numbers and standard operations on data types are defined (i.e. the basic notions defined in the file `Prelude.v`).

- `Reset Initial`

  This resets the system to its initial state where logical connectives, equality, booleans, natural numbers, standard operations on data types and specifications and elementary results about natural numbers are available (i.e. the basic notions defined in the files `Prelude.v`, `Specif.v`, `Peano.v` and `Wf.v`).

1.5.3 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.
```

1.5.4 Loading a file

- `Load` *filename*

  This loads the file `filename.v` in the current working directory. A system error occurs if the file does not exist.

  filename can contain `~`, `~userid`, and also environment variables, using, for instance, `$COQ` (in this case the path must be written between double-quotes).

1.5.5 Current path

The following commands are mostly self-explanatory:

- `Pwd` prints the current directory.
- `Cd dirname` changes the current directory.

1.5.6 Packages

A "Packages" facility, modeled after the similar notion in Gnu Emacs, is provided. It allows the user to divide his development in a modular way, to look for libraries in a user-controllable search path, and to describe dependencies among modules, while sharing common ancestors.

There is a global list of directories, the "load path", containing a list of all the system directories currently known by the system.

To add directories to this list from the command line of "coq", you use `-I <dir>`, like for camlc.

Now, there are several commands which manipulate this list:

```
Coq < AddPath dir.
```

Adds dir to the list. Remark that the current working directory is not present by default, but may be added by the command `AddPath "."`. The notations ".." and " " are similarly interpreted in their standard Unix way.

```
Coq < DelPath dir.
```

If dir is on the list, it is deleted.

```
Coq < Print LoadPath.
```

Prints out the load path.

There is also a component of the machine environment, which records the list of the "packages" which have been loaded into the environment previous to the declaration. You can think of this as a global variable, which is retracted by undo's and reset's.

When we do:

```
Coq < Require id [optional filename].
```

The system will search for `id.v` or `filename.v` in the loadpath, and if it finds it, it will do a Load of the file. In the file, there should be, as the last line, a command:

```
Coq < Provide id.
```

This will inform the system that the loaded file provided that "package"'s functionality.

So, for instance, I could do:

```
Coq < Require Prelude.
```

and if the file `Prelude.v` contained as a last line, `Provide Prelude`, then the fact that `Prelude` was loaded would be registered in the machine state. Then, if I do `Require Prelude` again, nothing happens, since it is already loaded.

If I do a `Reset All`, then the machine entry saying that `Prelude` is loaded will be popped, and I will have to load it again.

This means that instead of writing `Load Arith` in every file which uses `Arith`, and then having to comment it out, you can just write `Require Arith`, and then the system will take care of having `Arith` loaded only once.

```
Coq < Print Packages.
```

Prints out a list of the currently loaded packages.

1.5.7 State Manipulation Primitives.

```
Coq < Save State ident [optional string].
```

This saves the current machine state under the identifier. The optional string is a description of the state.

```
Coq < Print States.
```

This prints a list of the currently available states.

```
Coq < Remove State ident.
```

Removes the named state from the saved list. You cannot remove `Prelude` nor `Initial`.

```
Coq < Restore State ident.
```

Restores the named state to the machine.

```
Coq < Write States filename.
```

Writes out all the states in the save-list into the named file. The filename is appended with `.coq` automatically.

```
Coq < Read States filename.
```

Reads in the states from `filename.coq`, or if that doesn't exist, `filename`, and adds them to the current saved-list. If there are states with names which are the same as ones in the current saved-list, then the new ones are dropped on the floor. Hence, you can never overwrite the states `Initial` and `Prelude`, once they have been loaded into the image.

1.5.8 Quitting Coq

- `Quit`

  This command permits to quit Coq.

1.5.9 Recording mode: Open, Close

The commands given to the system Coq can be recorded in a file. For instance, if we give the command `Open demo` at the beginning of a sequence of commands and `Close` at the end, the file `demo.v` contains a recording of this sequence of commands.

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`.

1.5.10 Transcripts

Transcript mode and commands. The command, `Verify`, which takes a list of sequents, as printed by the `Show` command, and checks that the current goal-list is identical to them.

The command

```
Coq < Begin Transcript.
```

enables a mode in which whenever a "log" (via the `Open` command) is being kept, after each refining step, a `Verify` command is emitted, so that on reloading of the "log", the state of the machine is checked after every tactic step.

```
Coq < End Transcript.
```

turns off this mode.

Since this mode is only useful when you are logging commands to a file, obviously, if you do simply `Load foo`, then you don't get the logging, and hence no verify commands are inserted, either. So there is now a command:

```
Coq < Load Verbose filename.
```

which loads a file as if it were typed in from the keyboard. So if you do

```
Coq < Open "VERIF/Ack".
Coq < Load Verbose Ack.
      ... lots of output  ... (Oh, BOY, lots of output)
Coq < Close.
```

then the file `VERIF/Ack.v` will contain the "log" of "Ack.v", with Verify commands inserted.

This facility is extremely useful in two kinds of situations. First, when you are porting theory developments from one version of Coq to another one, which may contain slight incompatibilities. Print a transcript file of the development using the old version, and verify it with the new one.

The incompatibilities will emit an error message exactly at the place where the system response will differ, as opposed to sometimes many lines after, since a tactic may succeed locally. This will ease the portability problem in the future.

Second, you may find this facility useful when you add or subtract `Hints` commands from your development script, since the `Trivial` and `Auto` tactics may now have a different behavior.

1.6 The Coq Interface

A multi-window interface for the X-window system is included. In this section we assume you are running Coq with option "-x" in a Unix environment with the X11 library properly installed. 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.

1.6.1 Starting Up The Interface

The interface works on black and white, grayscale and color screens. To start up the interface, type:

```
Coq < Interface.
```

In order to specify a screen on which the interface should appear, the function "Interface" can take a string as argument corresponding to the machine name:

```
Coq < Interface "machine.domain.fr:0.0".
```

causes the interface windows to appear on the screen of the corresponding machine.

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.

1.6.2 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 Empty
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.

1.6.3 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 1.6.4 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 1.6.4.

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 Coq top level.

Using the interface or not does not change the internal state of the Coq engine. Parts of developments done under the X interface may be freely intermixed with parts done under the standart interation-loop.

If the window interface is started up again, it will return to the state it was in just before exiting.

1.6.4 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 _
Specialize _
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 _
```

Finally, two buttons control the facilities described in the section 2.3.

Realizer

```
Realizer
Realizer _
```

Program

```
Program
Program_all
Show Program
```

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 1.6.5.)

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.

1.6.5 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.

1.7 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.

1.7.1 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 the 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`.

1.7.2 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`, `(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: (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 (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 (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: (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).

1.7.3 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
        (* 0 *) [m:nat]True
      (* S k *) [k:nat][LE_funk:nat->Prop][m:nat]
            (<Prop>Match m with (* 0 *) False
                                (* S l *) [l:nat][LE_funkl:Prop](LE k l)))
```

We have `(LE_fun 0 m)` convertible with `True`, `(LE_fun (S n) 0)` 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) 0)->(LE_fun (S n) 0)` is convertible with `~(LE (S n) 0)`.

1.7.4 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:

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

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 correspond 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.

Generativity

Inductive types are equal up to renaming of their constructors. Indeed:

```
Coq < Inductive Set bit = Zero : bit | One : bit.
bit is inductively defined

Coq < Inductive Set color = Black : color | White :color.
color is inductively defined

Coq < Lemma bit_color_same_same : <bit>Zero=Black.

Coq < Proof (refl_equal bit Zero).
```

This equality under isomorphism is not always desirable, and is actually contrary to the standard convention of type equality in programming languages such as ML. One solution is to put "time-stamps" into inductive type definitions, so that each time we type one in, we get a "fresh" inductive type. This is the solution of most ML-like languages.

But we need something more flexible, since we can type in inductive types directly from the keyboard, e.g. when the conversion routines of the tactics library unfold constants which hide inductive types (e.g. when `nat` gets unfolded into `Ind(X:Set){X|X->X}`. Our solution is to put in user-specified "stamps".

Thus, if you want to "stamp" your inductive type `bit` to make it different from `color`, prefix the inductive definition command with the keyword `Generative`. For instance, the data types of `bool` and `nat` are declared in the prelude as being generative. Consequently:

```
Coq < Lemma ill_typed : <bool>true=Zero.
Illegal application : (eq bool true) cannot be applied to : Zero
Since the formal type : bool does not match the actual type : bit
Error Application would violate typings
```

For various purposes, though, one might want that two inductive types share the same "stamp", e.g. the inductive type `sumor` is generative, with definition:

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

and if we wanted to define an inductive type `sumorUV` (for some Set `U` and some Prop `V`), defining:

```
Generative Inductive Set sumorUV
    = inleft : U -> sumorUV | inright : V -> sumorUV.
```

would make a new generative inductive type, the members of which would *not* be convertible with member of the old one. Likewise, if we did not make the second definition generative, the same would hold (non-stamped types are always non-convertible with stamped types; otherwise, by the medium of a non-stamped type, two differently-stamped, but isomorphic, types, would be convertible). Thus, we need a way to say that the second definition has the same "stamp" as the first. We do with with the keyword `Upon`, which takes as argument the stamp:

```
Inductive Upon sumor Set sumorUV
    = inleft : U -> sumorUV | inright : V -> sumorUV.
```

And now, we will find that `sumorUV` is indeed convertible with `(sumor U V)`.

1.8 Proof-terms

1.8.1 Proof-terms

In Coq proofs can be manipulated as objects. We describe here the syntax of these proof-terms. These proofs-terms can be used by the tactic `Exact` when the proof of a subgoal is simple enough,

that it can be directly written as a term. When the proof of the initial goal is simple enough, instead of writing
`Theorem` *name.*
`Statement` *statement.*
`Goal.`
`Exact` *proof.*
`Save.`
we can write
`Theorem` *name.*
`Statement` *statement.*
`Proof` *proof.*

1.8.2 Syntax of proof-terms

Using an axiom

- If there is in the current context an already proved theorem, an already proved remark, an axiom or an hypothesis `u` of statement `P` then the symbol `u` is a proof of `P`.

Implication

- If `u` and `v` are proofs of the statements `P -> Q` and `P`, then `(u v)` is a proof of `Q`.
- If `u` is a proof of the statement `Q` in the current context extended with the declaration of an axiom `x` of statement `P` then `[x:P]u` is a proof of `P -> Q` in the current context.

Universal Quantification

- If `u` is a proof of the statement `(x:T)P` and `t` is a term of type `T`, then `(u t)` is a proof of $P[x \leftarrow t]$.
- If `u` is a proof of the statement `P` in the current context extended with the declaration of a variable `x` of type `T` then $[x : T]u$ is a proof of `(x:T)P` in the current context.

Conjunction

- If `u` and `v` are proofs of the statements `P` and `Q` then `(conj P Q u v)` is a proof of `P /\ Q`.
- If `u` is a proof of the statement `P /\ Q`, then `(proj1 P Q u)` is a proof of `P`. and `(proj2 P Q u)` is a proof of `Q`.

Disjunction

- If `u` is a proof of the statement `P`, then `(or_introl P Q u)` is a proof of `P\/Q`.
- If `v` is a proof of the statement `Q`, then `(or_intror P Q v)` is a proof of `P\/Q`.

- If u, v and w are proofs of statements (P \/ Q), (P -> C) and (Q -> C) then (or_ind P Q C u v w) is a proof of C.

Absurd and Negation

- If u is a proof of the statement False and P is any proposition then (False_ind P u) is a proof of P.
- If u and v are proofs of P and ~P then (v~u) is a proof of False.
- If u is a proof of False in the current context extended with the declaration of an axiom x of statement P, then [x:P]u is a proof of ~P.

  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 t is a term of type T and u is a proof of $P[x \leftarrow t]$ then (ex_intro T ([x:T]P) t u) is a proof of <T> Ex ([x:T]P).
- If u and v are proofs of the statements <T> Ex ([x:T]P) and (x:T)P->C where x does not appear free in C then (ex_ind T ([x:T]P) C v u) is a proof of C.

Equality

- If a is a term of type T then (refl_equal T a) is a proof of <T>a = a.
- If u and v are proofs of the statements (<A> a = b) and (P a) then (eq_ind A a P v b u) is a proof of (P b).

1.8.3 λ-terms as Proofs

In the previous section we have seen how to write proofs-terms. The notation given in each case 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.

Chapter 2

The System Coq as a Programming Language

In the first chapter, we showed how to write in Coq simple programs and prove their properties. In this chapter we adopt another point of view and use Coq as a language for the development of certified programs.

2.1 Introduction

This paradigm can be explained in two different ways.

First, we can think that we have a programming language in which we can express and prove logical properties. The programs are enriched with non-computational informations which justify the correctness of constructions. In the first part, we built a set of natural numbers, represented by structural properties. We can also, given a predicate `P` on natural numbers, which is represented as a term with type `nat->Prop`, build the set of natural numbers which satisfy `P`. In Coq it will be written `{x:nat|(P x)}`.

Second, we can think that we are doing constructive proofs. An intuitionistic proof admits a direct functional interpretation. It means that a proof of $\forall x.P(x) \Rightarrow \exists y.Q(x,y)$ can be translated into a function which takes as arguments an object x and a proof p of $P(x)$ and gives as value an object y and a proof of $Q(x,y)$. However, we want to be able to specify for instance that the expected program will only take x as input and gives y as output and that the proof of $P(x)$ is only needed to build a proof of $Q(x,y)$. For this purpose, Coq provides two sorts for propositions. The judgement `A:Prop` means that `A` is a logical property and that a proof `a` of `A` will not be used in a computational way. The judgement `A:Set` means that `A` is a computational proposition, also called *specification*. If `a` is a proof of `A`, then it represents the development of a program. Propositions and sets can be mixed.

From a proof of a specification, the system Coq extracts an "algorithm". More precisely, it removes the comments (proofs of logical propositions) from the proof. It produces typed programs but with no more dependent types. The section 2.2 gives definitions for useful specifications and develops examples of program development.

The extracted program is a skeleton of the proof of the specification. Consequentely, the proof and the extracted program share the same structure. When the expected extracted program is

known, it can be used as a guide for the development of a proof of a specification. A tactic has been written to automatize this process. A program can be associated to goal which is a specification. A tactic called **Program** develops the proof following this program and associate subprograms to generated subgoals. At the end, only subgoals corresponding to logical properties of programs are left to the user.

The language to express this program is called Real. Its syntax is analogous to the Coq syntax of terms. It contains abbreviations for useful constructions plus a mechanism to include annotations which are just Coq properties associated to subparts of the program. This functionality is explained in section 2.3. Finally, we want to really execute the programs developed in Coq and test them on consequent examples. The reductions involved in these programs are beta-reduction and pattern-matching, it is natural to see them as programs of a ML-like language. A small ML-language called Fml is integrated to the system Coq. In this language, we can define concrete types and write functions using general recursion and pattern-matching. In the Fml environment, only closed terms can be manipulated. They can be directly written or obtained by a translation of the programs extracted in the Coq proof environment. There is no compiler for Fml programs, instead, Coq provides various translators from Fml to existing ML compilers. The Fml language and toplevel is explained in section 2.4.

2.2 Development of programs with logical information

2.2.1 Motivations

Instead of having proofs which tell us about programs, we shall introduce 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.

2.2.2 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 `true` if there exists a proof of `A` and `false` 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 `true` if `<nat>n=(twice m)` and `false` 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 `(sigS2 A P Q)`.

2.2.3 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 developments of programs and will allow complete verification of the various proofs encountered.

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:

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

With instance `0` for `m` (tactic `Exists 0`), 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:

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

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

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

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 help to 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,true)
                      | (S n) -> let (m,p) = div2 n in
                                 match p with true  -> (m,false)
                                            | false -> ((S m),true)
```

We may specify another program for the division by two which will not use the boolean information. The specification of such a program is:

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

A proof of this specification can just be obtained by using the previous developed proof which introduces a boolean and then forgetting this boolean. But we can also build a proof which does not use this boolean.

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 an hypothesis :

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

The goal to prove is:

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

The hypothesis `H` is a logical proposition (of type `Prop`) and the goal to prove is an informative specification. Consequentely, it is forbidden to do an elimination on the hypothesis `H`. The logical statement `(<nat>n=(twice m))\/(<nat>n=(S (twice m)))` should just be considered as a comment and cannot be used to build a computational object.

This is unlike the previous development where the corresponding hypothesis `{<nat>n=(twice m)}+{<nat>n=(S (twice m))}` was an informative property, corresponding to the construction of a boolean value. The elimination was allowed and indeed, corresponded to a test on this boolean.

Actually, we do not need any test in order to provide a witness `(S m)` for the proof of the goal. Then, 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);;
```

2.2.4 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. We may represent this notion as 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:Set] : nat->Set
      = nil : (llist A 0)
      | cons : (n:nat)A->(llist A n)->(llist A (S n)).
```

The append function on such lists may be defined as:

```
Definition append = [A:Set][n,m:nat][l:(llist A n)][l':(llist A m)]
 (<[n:nat](llist A (plus n 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 (A:Set)(n,m:nat)(llist A n)->(llist A m)->(llist A (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. 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 A n1 (plus n2 n3) l1 (append A n2 n3 l2 l3)) admits for type the term (llist A (plus n1 (plus n2 n3))), but (append A (plus n1 n2) n3 (append A n1 n2 l1 l2) l3) admits for type (llist A (plus (plus n1 n2) n3)). These two types are not convertible to each other. A solution is to introduce a new notion of equality which compares l:(llist A n) and l':(llist A n'). This can be done with an inductive predicate:

```
Inductive Definition eqllist [A:Set; n:nat; l:(llist A n)]
       : (m:nat)(llist A m)->Prop
       = eql_refl : (eqllist A n l n l).
```

With this definition we may express the associativity of the append function as follows.

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

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

2.2.5 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 developments is that a proof of a 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 the 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 0 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) 0)` and `(LE (S n) (S m))->(LE n m)`.

2.2.6 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.

2.3 Developing certified programs

2.3.1 Motivations

We want to develop certified programs automatically proved by the system. That is to say, instead of giving a specification, an interactive proof and then extracting a program, the user gives the program he wants to prove and the corresponding specification. Using this information, an automatic proof is developped which solves the "informative" goals without the help of the user. When the proof is finished, the extracted program is guaranteed to be correct and corresponds to the one given by the user. The tactic uses the fact that the extracted program is a skeleton of its corresponding proof.

2.3.2 Syntax for tactics

The user has to give two things: the specification (given as usual by a goal) and the program (see 2.3.3). Then, this program is associated to the current goal (to know which specification it corresponds to) and the user can use different tactics to develop an automatic proof.

Realizer

First, the program is associated to the current goal by using the `Realizer` command. With this command, the program has to be given with the syntax indicated in part 2.3.3 and it is associated to the current goal.

Show Program

The command `Show Program` shows the program associated to the current goal. `Show Program n` shows the program associated to the nth subgoal.

Program

Then, an automatic process may be started. A program is associated to a goal by the user (for the initial goal) and by the tactic `Program` itself (for the subgoals). If not program is associated to the current goal, the tactic `Program` fails. This tactic generates a sequence of `Intro`, `Apply` or `Elim` tactics depending on the syntax of the program. For instance, if the program starts with a λ-abstraction, the `Intro` tactic is generated several times depending on the goal.

The `Program` tactic generates a list of subgoals which can be either logical or informative. Subprograms are associated to the informative subgoals.

Program_all

The `Program_all` tactic is equivalent to the following tactic: `Repeat (Program OrElse Auto)`. It repeats the `Program` tactic on every informative subgoal and try the `Auto` tactic on the logical subgoals. Note that the work of the `Program` tactic is considered finished when all the informative subgoals have been solved. This implies that logical lemmas can stay at the end of the automatic proof which have to be solved by the user.

2.3.3 Syntax for programs

Pure programs

The language to express programs is called Real*. Programs are explicitely typed† like terms extracted from proofs. Some extra expressions have been added to have a simpler syntax.

This is the raw form of what we call pure programs. But, in fact, it appeared that this simple type of programs is not sufficient. Indeed, all the logical part useful for the proof is not contained in these programs. That is why annotated programs are introduced.

Annotated programs

The notion of annotation introduces in a program a logical assertion that will be used for the proof. The aim of the `Program` tactic is to start from a specification and a program and to generate subgoals either logical or associated with programs. However, to find the good specification for subprograms is not at all trivial in general. For instance, if we have to find an invariant for a loop, or a well founded order in a recursive call.

So, annotations add in a program the logical part which is needed for the proof and which cannot be automatically retrieved. This allows the system to do proofs it could not do otherwise.

For this, a particular syntax is needed which is the following: since they are specifications, annotations follow the same internal syntax as Coq terms. We indicate they are annotations by putting them between `(:` and `:)`. Since annotations are Coq terms, they can involve abstractions over logical propositions that have to be declared. Annotated-λ have to be written between `[{` and `}]`. Annotated-λ can be seen like usual λ-bindings but concerning just annotations and not Coq programs.

Recursive Programs

Programs can be recursively defined using the syntax: `<`*type of the result*`>` `rec` *name of the induction hypothesis* `(:` *well-founded order of the recursion* `:)` and then the body of the program (see 2.3.5) which must always begin with an abstraction [x:A] where A is the type of the arguments of the fonction (also on which the ordering relation acts).

Abbreviations

Two abbreviations have been defined:

`<P>let (p:X;q:Y)=Q in S` is syntactic sugar for `<P>Match Q with [p:X][q:Y]S`

*It corresponds to F_ω plus inductive definitions

†This information is not strictly needed but was useful for type checking in a first experiment.

and
`<P>if B then Q else R` abbreviates matching on boolean expressions: `<P>Match B with Q R.`

Grammar

The grammar for programs is the following:

pg::= ident
| [x:pg]pg
| (x:pg)pg
| pg->pg
| (pg pg ... pg)
| (<pg>Match pg with pg-list)
| pg (: coqterm :)
| [{x:coqterm}]pg
| <pg>let $(x_1^1,\ldots,x_1^{k_1}:A_1;\ldots;x_n^1,\ldots,x_n^{k_n}:A_n)$ = pg in pg
| <pg>if pg then pg else pg
| <pg>rec identifier (: coqterm :) [x:pg]pg

The reference to an identifier of the Coq context (in particular a constant) inside a program of the language Real is a reference to its extracted contents.

2.3.4 Using the Interface

The interface can be used for developping automatic proofs. The different tactics are in the Proof Synthesis Windows. There are two buttons: Realizer and Program.

Realizer Tactic

`Realizer_`

The `Realizer_` button applies the `Realizer` tactic.

Program Tactics

`Program`
`Program_all`
`Show Program`

The `Program` button applies the `Program` tactic and so on.

2.3.5 Examples

Ackermann Function

Let us give the specification of Ackermann's function. We want to prove that for every n and m, there exists a p such that $ack(n,m) = p$ with:

$$ack(0,n) \quad = \quad n+1$$

$$\begin{aligned} ack(n+1,0) &= ack(n,1) \\ ack(n+1,m+1) &= ack(n,ack(n+1,m)) \end{aligned}$$

An ML program following this specification can be:

```
let rec ack = function
        0  -> (function m -> Sm)
      | Sn -> (function  0  -> ack n 1
                      | Sm -> ack n (ack Sn m))
```

Suppose we give the following definition in Coq of a ternary relation `(Ack n m p)` in a Prolog like form representing $p = ack(n,m)$:

```
Coq < 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).
```

Then the goal is to prove that $\forall n, m. \exists p. (Ack\ n\ m\ p)$, so the specification is: `(n,m:nat){p:nat|(Ack n m p)}`. The associated Real program corresponding to the above ML program will be:

```
Coq < Realizer [n:nat]
               (<nat->nat>Match n with
      (* 0 *) [m:nat](S m)
               (* S *) [y:nat][H:nat->nat][m:nat]
                         (<nat>Match m with (* 0 *) (H (S 0))
                                            (* S *) [m':nat][H':nat] (H H'))).
```

With the `Program_all` tactic, three logical lemmas are generated and are easily solved by using the properties Ack0, Ackn0 and AckSS.

```
3 subgoals
  (Ack 0 m (S m))
  ============================
    m : nat
    n : nat
subgoal 2 is:
  (Ack (S y) 0 x)
subgoal 3 is:
  (Ack (S y) (S y0) x0)
```

Euclidean Division

This example shows the use of *recursive programs*. Let us give the specification of the euclidean division algorithm. We want to prove that for a and b ($b > 0$), there exist q and r such that $a = b * q + r$ and $b > r$.

An ML program following this specification can be:

```
let div b a = divrec a where rec divrec = function
        if (b<=a) then let (q,r) = divrec (a-b) in (Sq,r)
                  else (0,a)
```

Suppose we give the following definition in Coq which describes what has to be proved, i.e., $\exists q \exists r.\ (a = b * q + r \wedge\ b > r)$:

```
Coq < Inductive Definition diveucl [a,b:nat] : Set
      = divex : (q,r:nat)(<nat>a=(plus (mult q b) r))->(gt b r)->(diveucl a b).
```

We assume that we have loaded the proper arithmetic libraries, using:

```
Coq < Require Arith.
Coq < Require Compare_dec.
Coq < Require Wf_nat.
```

The decidability of the ordering relation has to be proved first, by giving the associated function of type `nat->nat->bool`:

```
Coq < Theorem le_gt_dec : (n,m:nat){(le n m)}+{(gt n m)}.
Coq < Goal.
Coq < Realizer [n:nat](<nat->bool> Match n with
                      (* 0 *) [m:nat]true
                      (* S *) [n':nat][H:nat->bool][m:nat]
                              (<bool> Match m with
                               (* 0 *) false
                               (* S *) [m':nat][H':bool](H m'))).
Coq < Program_all.
Coq < Save.
```

Then the specification is `(b:nat)(gt b 0)->(a:nat)(diveucl a b)`. The associated program corresponding to the ML program will be:

```
Coq < Realizer
      [b:nat](<nat*nat>rec div (: lt :)
              [a:nat]
              (<nat*nat>if (le_gt_dec b a)
                        then (* le b a *)
                             <nat*nat>let (q,r:nat) = (div (minus a b))
                                      in <nat,nat>((S q),r)
                        else (* gt b a *) <nat,nat>(0,a))).
```

Where `lt` is the well-founded ordering relation defined by:

```
Coq < Definition lt = [n,m:nat](gt m n).
```

Note the syntax for recursive programs as explained before. The `rec` construction needs 4 arguments: the type result of the function (`nat*nat` because it returns two natural numbers) between `<` and `>`, the name of the induction hypothesis (which can be used for recursive calls), the ordering

relation `lt` (as an annotation because it is a specification), and the program itself which must begin with a λ-abstraction. The specification of `le_gt_dec` is known because it is a previous lemma. The term `(le_gt_dec b a)` is seen by the `Program` tactic as a term of type `bool` which satisfies the specification `{(le a b)}+{(gt a b)}`. The tactic `Program_all` can then be used, and the following logical lemmas are obtained:

```
2 subgoals
  (well_founded nat lt)
  ============================
    a0 : nat
    H : (gt b 0)
    b0 : nat
subgoal 2 is:
  <nat>x=(plus (mult (S q) b0) r)
```

Insertion sort

This example shows the use of *annotations*. Let us give the specification of a sorting algorithm. We want to prove that for a sorted list of natural numbers l and a natural number a, we can build another sorted list l', containing all the elements of l plus a.

An ML program implementing the insertion sort and following this specification can be:

```
let sort a l = sortrec l where rec sortrec = function
        []    -> [a]
      | b::l' -> if a<b then a::b::l' else b::(sortrec l')
```

Suppose we give the following definitions in Coq:

First, the decidability of the ordering relation:

```
Coq < Definition inf_dec: nat->nat->bool =
[n:nat](<nat->bool> Match n with
                [m:nat] true
                [n':nat][H:nat->bool][m:nat]
                        (<bool> Match m with
                                false
                                [m':nat][H':bool](H m'))).
```

The definition of the type `list_nat`:

```
Coq < Inductive Set list_nat = nil : list_nat | cons : nat -> list_nat -> list_nat.
```

We define the property for an element `x` to be *in* a list `l` as the smallest relation such that: $\forall a \forall l\ (In\ x\ l) \Rightarrow (In\ x\ (a :: l))$ and $\forall l\ (In\ x\ (x :: l))$.

```
Coq < Inductive Definition In [x:nat] : list_nat->Prop
= Inl  : (a:nat)(l:list_nat)(In x l) -> (In x (cons a l))
| Ineq : (l:list_nat)(In x (cons x l)).
```

A list t' is equivalent to a list t with one added element y iff: $(\forall x\ (In\ x\ t) \Rightarrow (In\ x\ t'))$ and $(In\ y\ t')$ and $\forall x\ (In\ x\ t') \Rightarrow ((In\ x\ t) \vee y = x)$. The following definition implements this ternary conjonction.

```
Coq < Inductive Definition equiv [y:nat;t,t':list_nat]: Prop =
      equiv_cons :
        ((x:nat)(In x t) -> (In x t'))
        -> (In y t')
        -> ((x:nat)(In x t') -> ((In x t) \/ <nat>y=x))
        -> (equiv y t t').
```

Definition of the property of list to be sorted, still defined inductively:

```
Coq < Inductive Definition sorted : list_nat->Prop
= sorted_nil  : (sorted nil)
| sorted_trans : (a:nat)(sorted (cons a nil))
| sorted_cons : (a,b:nat)(l:list_nat)(sorted (cons b l)) -> (le a b)
 -> (sorted (cons a (cons b l))).
```

Then the specification is:
`(a:nat)(l:list_nat)(sorted l)->{l':list_nat|(equiv a l l')&(sorted l')}.`

The associated Real program corresponding to the ML program will be:

```
Coq < Realizer [a:nat][l:list_nat](<list_nat>Match l with
  (cons a nil)
  [b:nat][m:list_nat][H:list_nat]
  <list_nat>if (inf_dec b a) (: {(le b a)}+{(gt b a)} :)
               then (cons b H)
               else (cons a (cons b m))).
```

Note that we have defined `inf_dec` as the program realizing the decidability of the ordering relation on natural numbers. But, it has no specification, so an annotation is needed to give this specification. This specification is used and then the decidability of the ordering relation on natural numbers has to be proved using the index program.

Suppose `Program_all` is used, a few logical lemmas are obtained (which have to be solved by the user):

```
7 subgoals
  (equiv a0 nil (cons a0 nil))
  ============================
    H : (sorted nil)
    l : list_nat
    a0 : nat
subgoal 2 is:
  (sorted (cons a0 nil))
subgoal 3 is:
  (equiv a0 (cons n y) (cons n x))
subgoal 4 is:
```

```
  (sorted (cons n x))
subgoal 5 is:
  (sorted y)
subgoal 6 is:
  (equiv a0 (cons n y) (cons a0 (cons n y)))
subgoal 7 is:
  (sorted (cons a0 (cons n y)))
```

Quicksort

This example shows the use of *programs using previous programs.* Let us give the specification of Quicksort. We want to prove that for a list of natural numbers l, we can build a sorted list l', which is a permutation of the previous one.

An ML program following this specification can be:

```
let rec quicksort l = function
   []    -> []
 | a::m -> let (l1,l2) = partition a m in
                let m1 = quicksort l1 and
                let m2 = quicksort l2 in m1@[a]@m2
```

Where `partition` is defined by:

```
let rec partition a l = function
[]    -> ([],[])
      | b::m -> let (l1,l2) = partition a m in
 if a<b then (l1,b::l2)
        else (b::l1,l2)
```

We now axiomatize lists over any ordered set A:

Declaration of the ordering relation:

```
Coq < Parameter A : Set.
Coq < Inductive Set list = nil : list | cons : A -> list -> list.
Coq < Inductive Definition In [x:A] : list->Prop
= Inl  : (a:A)(l:list)(In x l) -> (In x (cons a l))
| Ineq : (l:list)(In x (cons x l)).
Coq < Variable inf : A -> A -> Prop.
Coq < Definition sup = [x,y:A]~(inf x y).
Coq < Hypothesis inf_sup : (x,y:A){(inf x y)}+{(sup x y)}.
```

Definition of the concatenation of two lists:

```
Definition app = [l,m:list](<list>Match l with
       (* nil *) m
       (* cons *) [a:A][m:list](cons a))
   : list->list->list.
```

The auxiliary function `mil`:

```
Coq < Definition mil = [a:A][l,m:list](app l (cons a m)) : A->list->list->list.
```

Definition of the permutation of two lists:

```
Coq < Inductive Definition permut : list->list->Prop =
       permut_refl : (l:list)(permut l l)
     | permut_tran : (l,m,n:list)(permut l m)->(permut m n)->(permut l n)
     | permut_mil  : (a:A)(l,m:list)
                     (permut (cons a (app l m)) (app l (cons a m)))
     | permut_app  : (l,l',m,m':list)
                     (permut l l')->(permut m m')->(permut (app l m) (app l' m')).
```

The definitions `inf_list` and `sup_list` allow to know if an element is lower or greater than all the elements of a list:

```
Coq < Definition Rlist [R:A->Prop][l:list](a:A)(In a l)->(R a).
Coq < Definition inf_list [x:A](Rlist (inf x)).
Coq < Definition sup_list [x:A](Rlist (sup x)).
```

Definition of the property of a list to be sorted:

```
Coq < Inductive Definition sort : list->Prop =
       sort_nil : (sort nil)
     | sort_cons : (a:A)(l:list)(inf_list a l)->(sort l)->(sort (cons a l)).
```

Then the goal to prove is: $\forall l\ \exists m\ (sort\ m) \wedge (permut\ l\ m)$ and the corresponding specification is: `(l:list){m:list|(sort m)&(permut l m)}`. The associated Real program corresponding to the ML program will be:

```
Coq < Realizer
    (<list>rec quick (: ltl :)
           [l:list]
           (<list>Match l with
            (* nil *)  nil
           (* cons *)  [a:A][m:list][t:list]
                       <list>let (l1,l2:list) = (Partition a m) in
                       (mil a (quick l1) (quick l2)))).
```

Where `ltl` is the well-founded ordering relation defined by:

```
Definition length = [l:list](<nat>Match l with (* nil *) 0
                              (* cons a m *) [a:A][m:list]S).
Definition ltl = [l,m:list](gt (length m) (length l)).
```

And `Partition` is the definition corresponding to $\exists l_1 \exists l_2.\ (sup\_list\ a\ l_1) \wedge (inf\_list\ a\ l_2) \wedge (l \equiv l_1 @ l_2) \wedge (ltl\ l_1\ (a :: l)) \wedge (ltl\ l2\ (a :: l))$:

```
Coq < Inductive Set Partition_spec [a:A; l:list] =
      Split_intro : (l1,l2:list)(sup_list a l1)->(inf_list a l2)
                    ->(permut l (app l1 l2))
```

```
                    ->(ltl l1 (cons a l))->(ltl l2 (cons a l))
                    ->(Partition_spec a l).

Coq < Theorem Partition : (a:A)(l:list)(Partition_spec a l).
Coq < Goal.
Coq < Realizer [a:A][l:list]
   (<list*list> Match l with
    (* nil *) <list,list>(nil,nil)
  (* cons  *) [b:A][m:list][l1:list*list]
              (<list*list>let (l1,l2:list) = l1 in
                 (<list*list>if (inf_sup a b)
                             then (* inf a b *) <list,list>(l1,(cons b l2))
     else (* sup a b *) <list,list>((cons b l1),l2)))).
Program_all.
Simpl; Auto.
Save.
```

Then `Program_all` gives the following logical lemmas (they have to be resolved by the user):

```
3 subgoals
  (well_founded list ltl)
  ============================
    l : list
subgoal 2 is:
  (sort (mil a x1 x0))
subgoal 3 is:
  (permut (cons a y) (mil a x1 x0))
```

2.4 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.

2.4.1 Sketching the extraction algorithm

The main ideas have already been exposed previously in section 2.2.2. 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}).

2.4.2 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 2.4.10. Yet in practice, this happens very rarely, especially in the case of proofs of actual program specifications.

2.4.3 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. The code extracted for the CAML language can be compiled by CAML Light compiler.
- LML, The lazy ML implementation of Göteborg [‡].

[‡]The LML compiler is available for free by anonymous ftp at the Chalmers university in 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.

2.4.4 Using Fml

Fml has its own toplevel, which is obtained by typing `Fml` from the vernacular toplevel. 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:

```
Coq < Fml.

Fml < let a = 1.
a is defined:
1
Fml < let B = (1*a)-3.
B is defined:
((1*a)-3)
Fml < Print B.
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.*

[§]This implementation is still under development. Interested readers may contact *maranget@margaux.inria.fr*

```
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 = 0 | S NAT.
type NAT is defined:
 == inductive
    0 | S NAT
```

which means that the type `NAT` has two constructors `0` 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.

2.4.5 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 2.4.9). 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 toplevel, the user may want to see the extracted terms corresponding to his proofs. He can do so using the following commands (which is available at the Coq and not at the Fml level):

- `Extraction` : prints the whole extracted context in his F_ω form; i.e. the typed counterpart of the Fml program.

- `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`.

2.4.6 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*.

2.4.7 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 Fml commands to switch the flag are :

```
Fml < Lml.
Fml < Gaml.
```

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 `Optimize` ; it tries to evaluate the definitions present in the current environement and possibly expands and deletes the non-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.

2.4.8 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.

2.4.9 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. For a better understanding of these features, we advise the reader to take a look at the different examples given later on.

2.4.10 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*. To swich off the CAML Light type checker, use the function `obj__magic` 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` [¶].

2.5 Some examples

2.5.1 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 *0* and *S*. The corresponding Fml definition would be:

```
Inductive nat = O | S nat.
```

To use the file, we begin by loading it into the Coq environment:

```
Coq < AddPath "$COQTH/PROGRAMS".
Coq < Require Euclid_proof.
```

Once this is done, we can play with the extracted program:

```
Coq < Fml.

Fml < Extract.
Fml < Print eucl_dev.

eucl_dev = [b][a](gt_wf_ind a
             [n]
              [H0]
               (sumbool_rec
                   (diveucl_rec (minus n b) b [q][r](divex n b (S q) r)
                   (H0 (minus n b)))
```

[¶]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*

```
                (divex n b O n)
               (le_gt_dec b n)))

Fml < Types.
        .....
diveucl  == ( ( prod  nat)  nat)
```

The function *eucl_dev* will take two unary numbers as arguments, and return a 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 -> O
Fml <   | (S m) -> 1+(to_int m)
Fml <  end match.

Fml < let rec to_nat = [N]
Fml <   (if N=O then O else (S (to_nat (N-1)))).

Fml < let eucl_int = [N,M]
Fml <   (match (eucl_dev (to_nat N) (to_nat M)) with
Fml <     (pair q r) -> (pair (to_int q)(to_int r))
Fml <   end match).

Fml < Write CAML File euclid.

Fml < Drop.
Coq < Quit.
```

To run the CAML program, we can use the camllight compiler.

```
unix> camllight
>       Caml Light version 0.5

#include "euclid";;
#eucl_int 3 3;;
- : (int, int) prod = pair_C (1, 0)
#eucl_int 150 32131;;
- : (int, int) prod = pair_C (214, 31)
```

2.5.2 Heapsort

The file (`Heap_proof.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:

```
Coq < AddPath "$COQTH/LISTS".
Coq < Require List.
Coq < AddPath "$COQTH/PROGRAMS".
Coq < Require Heap_proof.
```

As we saw in 2.4.9, we have to instantiate or realize by hand some of the Coq variables.

```
Coq < Fml.
Fml < Free Vars.

*** [False_rec :(P:Data)P]

*** [eq_rec :(A:Data)A->(P:Data)P->A->P]

*** [Acc_rec :(A:Data)(P:Data)(A->(A->P)->P)->A->P]

*** [A :Data]

*** [inf_total :A->A->sumbool]

*** [carac :A->A->nat]
```

Note that the sort **Set** is mapped to the sort **Data** in the world of realizers.

The variables *False_rec*, *eq_rec* and *Acc_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.
```

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 < Write CAML File heap.
```

Now, with CAML Light for instance, we can test our program.

```
#include "heap";;
#heapsort (inter 3000);;
cons_C (1, cons_C (2, cons_C (3, cons_C (4,(cons_C (50, ...
```

This shows the program is quite efficient for CAML code.

2.5.3 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:

$$[[1]\ ;\ [3]\ ;\ [6];\ [2]\ ;[4];[9]] \mapsto [[1;3];[2;6];[4;9]]$$
$$[[1;3];\ [2;6];\ [4;9]] \mapsto [[1;2;3;6];[4;9]]$$
$$[[1;2;3;6];\ [4;9]] \mapsto [[1;2;3;4;6;9]]$$

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:Data)(P:Data)A->(A->(A->P)->P)->P.
```

This axiom is preloaded in the environment and realized by the **Fml** program:

```
[f,x](let rec F = [y](f y F) in (F x)).
```

So `Mergesort.v` will be used exactly as `Heap.v`. The computational behavior of the resulting program is quite satisfactory and can be compared with hand-written code:

```
#mergesort (inter 3000);;
(cons_C
 (1,
  (cons_C
   (2,
    (cons_C
```

```
    (3,
     (cons_C
      (4,
       (cons_C
....
```

2.5.4 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 `Peano.v` contains a basic axiomatization of natural numbers.
- The file `Wf.v` concerns well-founded induction. These files are automatically loaded in the initial state of the system.

```
(*************************************************************************)
(*          Projet Formel - Calculus of Inductive Constructions V5.8     *)
(*************************************************************************)
(*                                                                       *)
(*          Prelude :  Logical connectives, quantifiers, equality        *)
(*                                                                       *)
(*************************************************************************)

Chapter Prelude.

Section Logic.

Inductive Definition True : Prop = I : True.

Section Negation.

(* Absurdity *)

Inductive Definition False : Prop = .

(* Negation *)

Definition not [A:Prop]A->False.
 Syntax not "~_".

End Negation.

Section Conjunction.

(* Pairing / Introduction *)

 Syntax and "_/\\_".

Syntax conj "<_,_>{_,_}".
Inductive Definition and [A,B:Prop] : Prop = conj : A->B->(A/\B).

Section Projections.

Variables A,B : Prop.

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

End Disjunction.

(* Equivalence *)

Definition iff = [P,Q:Prop](P -> Q) /\ (Q -> P).
Syntax iff "_ <-> _".

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.
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.

Theorem absurd : (A:Prop)(C:Prop)A->(~A)->C.
Goal.
Unfold not; Intros A C h1 h2.
(*    h2 : A->False
      h1 : A
      C : Prop
      A : Prop
      subgoal  C *)
Elim (h2 h1).
Save.

Section equality.
 Variable A,B : Set.
 Variable f   : A->B.
 Variable x,y,z : A.

 Theorem sym_equal : (<A>x=y) -> <A>y=x.
 Goal.
 Intros h; Elim h.
 (*    h : <A>x=y
       subgoal  <A>x=x *)
 Apply refl_equal.
 Save.

 Theorem trans_equal : (<A>x=y) -> (<A>y=z) -> <A>x=z.
```

```
 Goal.
 Intros h1 h2; Elim h2; Apply h1.
 Save.

 Theorem f_equal : (<A>x=y)-><B>(f x)=(f y).
 Goal.
 Intros h; Elim h.
 (*    h : <A>x=y
       subgoal  <B>(f x)=(f x) *)
 Apply refl_equal.
 Save.

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.

Provide Prelude.
```

```
(*****************************************************************************)
(*          Projet Formel - Calculus of Inductive Constructions V5.8         *)
(*****************************************************************************)
(*                                                                           *)
(*     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}).


(**********)
(* Choice *)
(**********)

Lemma Choice : (S,S':Set)(R:S->S'->Prop)((x:S){y:S'|(R x y)})
                      -> {f:S->S'|(z:S)(R z (f z))}.
Goal.
Intros S S' R H.
Exists [z:S](<S'>Match (H z) with [y:S'][h:(R z y)]y).
Intro z; Elim (H z); Trivial.
Save.

Lemma Choice2 : (S,S':Set)(R:S->S'->Set)((x:S){y:S' & (R x y)})
                      -> {f:S->S' & (z:S)(R z (f z))}.
Goal.
Intros S S' R H.
Exists [z:S](<S'>Match (H z) with [y:S'][h:(R z y)]y).
Intro z; Elim (H z); Trivial.
Save.

Lemma bool_choice : (S:Set)(R1,R2:S->Prop)((x:S){(R1 x)}+{(R2 x)}) ->
 {f:S->bool|(x:S)( (<bool>(f x)=true  /\ (R1 x))
```

```
              \/ (<bool>(f x)=false /\ (R2 x)))}.
Goal.
Intros S R1 R2 H.
Exists [z:S](<bool>Match (H z) with [r:(R1 z)]true [r:(R2 z)]false).
Intro z; Elim (H z); Auto.
Save.


(*******************************)
(* Self realizing propositions *)
(*******************************)

Axiom False_rec : (P:Set)False->P.
Definition except = False_rec. (* for compatibility with previous versions *)


Theorem absurd_set : (A:Prop)(C:Set)A->(~A)->C.
Goal.
Intros A C h1 h2.
(*    h2 : ~A
      h1 : A
      C : Prop
      A : Prop
      subgoal  C *)
Apply False_rec.
Apply (h2 h1).
Save.

Theorem and_rec : (A,B:Prop)(C:Set)(A->B->C)->(A/\B)->C.
Goal.
Intros A B C F AB; Apply F; Elim AB; Auto.
Save.

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.

Provide Specif.
```

```
(*************************************************************************)
(*          Projet Formel - Calculus of Inductive Constructions V5.8     *)
(*************************************************************************)
(*                                                                       *)
(*                           Natural numbers                             *)
(*                            Peano Axioms                               *)
(*                                                                       *)
(*************************************************************************)

Theorem eq_S : (n,m:nat)(<nat>n=m)->(<nat>(S n)=(S m)).
Goal.
Intros n m H ; Apply (f_equal nat) ; Auto.
Save.
Hint eq_S.

(* The predecessor function *)

Definition pred : nat->nat
     = [n:nat](<nat>Match n with (* O *) O
                                 (* S u *) [u,v:nat]u).

Theorem pred_Sn : (m:nat)<nat>m=(pred (S m)).
Goal.
Auto.
Save.

Theorem eq_add_S : (n,m:nat)(<nat>(S n)=(S m))->(<nat>n=m).
Goal.
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.
Immediate eq_add_S.

(* A consequence of the previous axioms *)

Theorem not_eq_S : (n,m:nat)(~<nat>n=m)->~<nat>(S n)=(S m).
Goal.
Red; Intros n m H1 H2 ; Apply H1; Auto.
Save.
Hint not_eq_S.

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

Theorem O_S : (n:nat)~(<nat>O=(S n)).
Goal.
Red ; Intros n H.
(*  False
    ============================
```

```
      H : <nat>O=(S n)
      n : nat *)
Change (IsSucc O) ; Rewrite H ; Simpl ; Auto.
Save.
Hint O_S.

Theorem n_Sn : (n:nat)~<nat>n=(S n).
Goal.
Induction n ; Auto.
Save.
Hint n_Sn.

(*************************************************)
(*      Addition                                 *)
(*************************************************)

Definition plus = [n,m:nat](<nat>Match n with
      (* O *) m
    (* S p *) [p,plus_p_m:nat](S plus_p_m)).

Lemma plus_n_O : (n:nat)<nat>n=(plus n O).
Goal.
Induction n ; Simpl ; Auto.
Save.
Hint plus_n_O.

Lemma plus_n_Sm : (n,m:nat) <nat>(S (plus n m))=(plus n (S m)).
Goal.
Intros m n; Elim m; Simpl; Auto.
Save.
Hint plus_n_Sm.

(***************************************)
(*      Multiplication                 *)
(***************************************)

Definition  mult =
   [n,m:nat](<nat> Match n with (* O *) O
                            (* S p *) [p:nat](plus m)).

Goal (n:nat)<nat>O=(mult n O).
Induction n; Simpl; Auto.
Save mult_n_O.

Goal (n,m:nat)<nat>(plus (mult n m) n)=(mult n (S m)).
Intros; Elim n; Simpl; Auto.
Intros p H; Rewrite H; Elim plus_n_Sm; Apply eq_S.
Pattern 1 3 m; Elim m; Simpl; Auto.
Save mult_n_Sm.

(*********************************************************************)
(* Definition of the usual orders, the basic properties of le and lt   *)
(* can be found in files Le and Lt                                     *)
(*********************************************************************)
```

```
(* An inductive definition to define the order *)

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.

Definition lt [n,m:nat](le (S n) m).
Hint Unfold lt.

Definition ge [n,m:nat](le m n).
Hint Unfold ge.

Definition gt [n,m:nat](lt m n).
Hint Unfold gt.

(*********************************************************)
(* Pattern Matching on natural numbers                   *)
(*********************************************************)

Theorem nat_case : (n:nat)(P:nat->Prop)(P O)->((m:nat)(P (S m)))->(P n).
Goal.
Intros n P H H0 ; Elim n ; Auto.
Save.

(*********************************************************)
(* Principle of double induction                         *)
(*********************************************************)

Theorem nat_double_ind : (R:nat->nat->Prop)
     ((n:nat)(R O n)) -> ((n:nat)(R (S n) O))
     -> ((n,m:nat)(R n m)->(R (S n) (S m)))
     -> (n,m:nat)(R n m).
Goal.
Induction n; Trivial.
Induction m; Auto.
Save.

Provide Peano.
```

```
(*************************************************************************)
(*          Projet Formel - Calculus of Inductive Constructions V5.8     *)
(*************************************************************************)
(*                                                                       *)
(*                        Well-founded recursion                         *)
(*                                                                       *)
(*************************************************************************)
(*      Definitions and lemmas concerning well-founded induction         *)
(*************************************************************************)

Chapter  Well_founded.

Variable A : Set.

(* The accessibility predicate is defined to be non-informative *)

Inductive Definition Acc [R:A->A->Prop] : A -> Prop
   = Acc_intro : (x:A)((y:A)(R y x)->(Acc R y))->(Acc R x).

(* the informative elimination :
   to be realised by a fixpoint
   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).

(* A relation is well-founded if every element is accessible *)

Definition well_founded = [R:A->A->Prop](a:A)(Acc R a).

(* well-founded induction *)

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.

End Well_founded.

Provide Wf.
```

Index

Unité de Recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399