The Coq Proof Assistant, Reference Manual, Version 5.10

Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Gérard Huet, Pascal Manoury, César Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, et al.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

The Coq Proof Assistant Reference Manual

Version 5.10

Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Gérard Huet, Pascal Manoury, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, Benjamin Werner

N° 0177

July 1995

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

Rapport technique

ISSN 0249-6399

The Coq Proof Assistant
Reference Manual

Version 5.10 *

Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Gérard Huet, Pascal Manoury, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, Benjamin Werner

Projet Coq

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

*This research was partly supported by ESPRIT Basic Research Action "Types" and by the GDR "Programmation" co-financed by MRE-PRC and CNRS.

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

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

1$^{\text{st}}$ Printing July 1$^{\text{st}}$, 1995

Contents

Credits

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 Coq 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 development 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 objects, in the tradition of Bishop's constructive mathematics, is based on a realizability 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 released in 1991.

Coq was ported to the new implementation Caml-light of X. Leroy and D. Doligez by D. de Rauglaudre (Version 5.7) in 1992. A new version of Coq was then coordinated by C. Murthy, with 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. This system (Version 5.8) was released in May 1993. A Centaur interface CTCoq was then developed by Y. Bertot from the Croap project from INRIA-Sophia-Antipolis.

In parallel, G. Dowek and H. Herbelin developed a new proof engine, allowing the general manipulation of existential variables consistently with dependent types in an experimental version of Coq (V5.9).

The present version V5.10 of Coq is based on a generic system for manipulating terms with binding operators due to Chet Murthy. A new proof engine allows the parallel development of partial proofs for independent subgoals. The structure of these proof trees is a mixed representation of derivation trees for the Calculus of Inductive Constructions with abstract syntax trees for the tactics scripts, allowing the navigation in a proof at various levels of details. The proof engine allows generic environment items managed in an object-oriented way. This new architecture, due to C. Murthy, supports several new facilities which make the system easier to extend and to scale up:

- User-programmable tactics are allowed
- It is possible to separately verify development modules, and to load their compiled images without verifying them again - a quick relocation process allows their fast loading
- A generic parsing scheme allows user-definable notations, with a symmetric table-driven pretty-printer
- Syntactic definitions allow convenient abbreviations
- A limited facility of meta-variables allows the automatic synthesis of certain type expressions, allowing generic notations for e.g. equality, pairing, and existential quantification.

In the Fall of 1994, C. Paulin-Mohring replaced the structure of inductively defined types and families by a new structure, allowing the mutually recursive definitions. P. Manoury implemented a translation of recursive definitions into the primitive recursive style imposed by the internal recursion operators, in the style of the ProPre system. C. Muñoz implemented a decision procedure for intuitionistic propositional logic, based on results of R. Dyckhoff. J.C. Filliâtre implemented a decision procedure for first-order logic without contraction, based on results of J. Ketonen and R. Weyhrauch. Finally C. Murthy implemented a library of inversion tactics, relieving the user from tedious definitions of "inversion predicates".

Rocquencourt, Feb. 1st 1995
Gérard Huet

Chapter 1

Introduction

This document is the Reference Manual of version V5.10 of the Coq proof assistant. A companion volume, the Coq Tutorial, is provided for the beginners. It is advised to read the Tutorial first. Additional documentation is described in chapter 15.

All services of the Coq proof assistant are accessible by interpretation of a command language. A command is a string ended with a period.

Coq has an interactive mode in which commands are interpreted as the user types them in from the keyboard and a compiled mode where commands are processed from a file. Other modes of interaction with Coq are possible, through an emacs shell window, or through a customized interface with the Centaur environment (CTCoq). These facilities are not documented here.

- The interactive mode may be used as a debugging mode in which the user can develop his theories and proofs step by step, backtracking if needed and so on. The interactive mode is run with the `coqtop` command from the operating system (which we shall assume to be some variety of UNIX in the rest of this document).
- The compiled mode acts as a proof checker taking a file containing a whole development in order to ensure its correctness. Moreover, Coq's compiler provides an output file containing a compact representation of its input. The compiled mode is run with the `coqc` command from the operating system. Its use is documented in chapter 13.

Coq offers two kinds of services : logical services and operating services.
We divide the logical services in two main parts :

- a specification language in which the user axiomatizes his own theories. This specification language is known as Gallina which mainly provides declaration and definition mechanisms. It is documented in chapter 2.
- a proof editing mode providing tools for proof development. Proofs services are again of two kinds :
  - proofs pragmas such as switching on and off the proof editor, restarting a proof, etc ... They are documented in chapter 3.
  - tactics which are the implementation of logical reasoning steps. The whole chapter 4 is devoted to their documentation.

For a more fundamental understanding of the logical framework, we urge the user of `Coq` to read chapter 6.

The so-called operating services are :

- a file system service including modules facilities
- displaying features
- user's syntax handling
- miscellaneous pragmas

They are documented in chapter 5.

Notations In the rest of this document, `Coq`'s grammar terminals will be written in `typewriter` font. Non-terminals are

1. *Fwterm* which denotes an F_ω term (see section 5.6.6).
2. *ident* which denotes an *identifier* in the usual sense. Characters such as `_` and `'` are allowed to appear in identifiers, besides alpha-numerical characters.
3. *num* which denotes a positive natural number (*e.g.* a sequence of digits with no blanks).
4. *pattern* which denotes any CIC-term belonging to a restricted class (see section 2.5.5).
5. *ref* which is either an *ident* or a *num*.
6. *string* which denotes any sequence of characters enclosed between two `"`.
7. *tac* which denotes any simple tactic's name or composed tactical (see section 4.11).
8. *term* which denotes any CIC-term (see section 2.2).
9. *sort* which denotes one of the special CIC-constants called a sort (see section 6.1.1).

Chapter 2

The Gallina specification language

2.1 Lexical conventions

Blanks Space, newline and horizontal tabulation are considered as blanks. Blanks are ignored but they separate tokens.

Comments Comments in Coq are enclosed between (* and *), and can be nested. Comments are treated as blanks.

Identifiers Identifiers are sequences of letters, digits, _, $ and ', that do not start with a digit or '. That is, they are recognized by the following regular expression

ident ::= (a..z|A..Z|_|$){a..z|A..Z|0..9|_|$|'}+

Identifiers can contain at most 80 characters, and all characters are meaningful.

Integers Integers are sequences of digits, optionally preceded by a minus sign, that is

integer ::= [-]{0..9}+

Strings Strings are delimited by " (double quote), and enclosed a sequence of any characters different from " and \, or one of the following sequences

Sequence	Character denoted
\\	backslash (\)
\"	double quote (")
\n	newline (LF)
\r	return (CR)
\t	horizontal tabulation (TAB)
\b	backspace (BS)
ddd	the character with ASCII code *ddd* in decimal

Strings can be split on several lines using a backslash (\) at the end of each line, just before the newline. For instance,

```
Coq < AddPath "$COQTOP/\
Coq < contrib/Rocq/LAMBDA".
```

is correctly parsed, and equivalent to

```
Coq < AddPath "$COQTOP/contrib/Rocq/LAMBDA".
```

Keywords The following identifiers are reserved keywords, and cannot be employed otherwise:

Definition	DelPath	Dependent	end	Grammar
in	Load	LoadPath	NewSyntax	Non
Orelse	Proof	Qed	Quit	Remove
Reset	Restore	State	Syntax	with

Although they are not considered as keywords, it is inadvisable to use words of the following list as identifiers:

Abort	Abstraction	All	Axiom	Begin
Cd	Chapter	Check	CheckGuard	CoFixpoint
Compute	Defined	Definition	Drop	Elimination
End	Eval	Explain	Extraction	Fact
Fixpoint	Focus	for	Go	Goal
Hint	Hypothesis	Immediate	Induction	Inductive
Infix	Inspect	Lemma	Let	Local
Minimality	ML	Module	Modules	Mutual
Node	Opaque	Parameter	Parameters	Print
Proofs	Prop	Pwd	Qed	Remark
Require	Restart	Resume	Save	Scheme
Script	Search	Section	Set	Show
Silent	States	Suspend	Syntactic	Theorem
Token	Transparent	Tree	Type	TypeSet
Undo	Unfocus	Variable	Variables	Write

Other keywords and user's tokens The following sequences of characters are also keywords:

```
|   :   :=   =   >   >>   <>
<<  <   ->   ;   #   *    ,
?   @   ::   /   <-
```

You can add new tokens with the command `Token` (see section 5.7.4). New tokens must be sequences, without blanks, of characters taken from the following list:

```
< > / \ - + = ; , | ! @ # % ^ & ? * : ~ $ _ a..z A..Z ' 0..9
```

that do not start with a character from

```
$ _ a..z A..Z ' 0..9
```

Lexical ambiguities are resolved according to the "longest match" rule: when a sequence of the previous characters can be decomposed into several different ways, then the first token is the longest possible one (among all tokens defined at this moment), and so on.

2.2 Basic syntax of terms

The basic set of terms form the *Calculus of Inductive Constructions* also called CIC. The formal presentation of CIC is given in chapter 6. We give here (an approximation of) the syntax available in Coq.

term	::=	*ident*
	|	*sort*
	|	(*binder*) *term*
	|	[*binder*] *term*
	|	[*lident*] *term*
	|	[*ident* = *term*] *term*
	|	(*terms*)
	|	< *term* >`Match` *term* `with` *terms* `end`
	|	< *term* >`Case` *term* `of` *terms* `end`
	|	`Fix` *ident* { *fixdecls* }
	|	< *term* >`let` (*letdecls*) = *term* `in` *term*
	|	< *term* >`let` (*lident*) = *term* `in` *term*
	|	< *term* >`if` *term* `then` *term* `else` *term*
terms	::=	*term*
	|	*term terms*
lident	::=	*ident*
	|	*ident* , *lident*
binder	::=	*lident* : *term*
sort	::=	`Prop`
	|	`Set`
	|	`Type`
	|	`Typeset`
fixdecls	::=	*fixdecl*
	|	*fixdecl* `with` *fixdecls*
fixdecl	::=	*ident* / *num* : *term* := *term*
letdecls	::=	*binder*
	|	*binder* ; *letdecls*

Remarks :

1. (*terms*) associates to the left.
2. The syntax [*lident*] *term* allows not to give types in abstractions.
3. The syntax [*ident* = *term*] *term* allows to define a β-redex.

   Example : $[x{=}T_1]T_2$ is equivalent to $([x]T_2\ T_1)$.
4. The syntax < *term* >`let` (*letdecls*) = *term* `in` *term* is a macro for a `Match` or a `Case` with one only case.

   Variants : The syntax < *term* >`let` (*lident*) = *term* `in` *term* is a variant of a precedent but types are not needed in *lident*.

5. The syntax < *term* >`if` *term* `then` *term* `else` *term* is a macro for `Match` or a `Case` with two only cases.

2.3 Declarations

The declaration mechanism allows the user to specify his own basic objects. Declared objects play the role of axioms or parameters in mathematics. A declared object is an *ident* associated to a *term*. A declaration is accepted by `Coq` iff this *term* is a well-typed specification in the current context of the declaration. This *term* is considered to be the type, or specification, of the *ident*.

2.3.1 `Axiom` *ident* : *term*.

This command links *term* to the name *ident* as its specification in the global context. The fact asserted by *term* is thus assumed as a postulate.

Error message :

1. `Clash with previous constant` *ident*

Variants :

1. `Parameter` *ident* : *term*.
   Is equivalent to `Axiom` *ident* : *term*
2. `Parameters` *lident* : *term*.
   Links *term* to the names comprising the list *lident*

2.3.2 `Variable` *ident* : *term*.

This command links *term* to the name *ident* in the context of the current section. The name *ident* will be unknown when the current section will be closed. One says that the variable is *discharged*. Using the `Variable` command out of any section is equivalent to `Axiom`.

Error message :

1. `Clash with previous constant` *ident*

Variants :

1. `Variables` *lident* : *term*.
   Links *term* to the names comprising the list *lident*
2. `Hypothesis` *lident* : *term*.
   Is equivalent to `Variables` *lident* : *term*

See also : section 2.6

It is advised to use the keywords `Axiom` and `Hypothesis` for logical postulates (i.e. when the assertion *term* is of sort `Prop`), and to use the keywords `Parameter` and `Variable` in other cases (corresponding to the declaration of an abstract mathematical entity).

2.4 Definitions

Definitions differ from declarations since they allow to give a name to a term whereas declarations were just giving a type to a name. That is to say that the name of a defined object can be replaced at any time by its definition. This replacement is called δ-conversion (see section 6.3). A defined object is accepted by the system iff the defining term is well-typed in the current context of the definition. Then the type of the name is the type of term. The defined name is called a *constant* and one says that *the constant is added to the environment*.

A formal presentation of constants and environment is given in section 6.4.

2.4.1 `Definition` *ident* `:=` *term*`.`

This command binds the value *term* to the name *ident* in the environment, providing that *term* is well-typed.

Error message :

1. `Clash with previous constant` *ident*

Variants :

1. `Definition` *ident* `:` $term_1$ `:=` $term_2$`.` It checks that the type of $term_2$ is definitionally equal to $term_1$, and registers *ident* as being of type $term_1$, and bound to value $term_2$.

Error message :

1. `In environment the term:` $term_2$ `does not have type` $term_1$`. Actually, it has type` $term_3$.

See also : sections 5.6.2, 4.4.4

2.4.2 `Local` *ident* `:=` *term*`.`

This command binds the value *term* to the name *ident* in the environment of the current section. The name *ident* will be unknown when the current section will be closed and all occurrences of *ident* in persistent objects (such as theorems) defined within the section will be replaced by *term*. One can say that the `Local` definition is a kind of *macro*.

Error message :

1. `Clash with previous constant` *ident*

Variants :

1. `Local` *ident* `:` $term_1$ `:=` $term_2$`.`
   Checks that the type of $term_2$ is definitionally equal to $term_1$, and registers *ident* as being of type $term_1$.

See also : sections 2.6, 5.6.2, 4.4.4

2.5 Inductive definitions

This version of Coq contains a new implementation of inductive definitions. It is formally presented in section 6.5.

2.5.1 `Inductive` *ident* `:` *term* `:=` $ident_1$ `:` $term_1$ `| ..` `|` $ident_n$ `:` $term_n$ `.`

This command is used to define inductive types and inductive families such as inductively defined relations. The name *ident* is the name of the inductively defined object and *term* is its type. The names $ident_1$, .., $ident_n$ are the names of its constructors and $term_1$, .., $term_n$ their respective types. The types of the constructors have to satisfy a *positivity condition* (see section 6.5.3) for *ident*. This condition ensures the well-foundedness of the inductive definition. If this is the case, the constants *ident*, $ident_1$, .., $ident_n$ are added to the environment with their respective types. According to the arity of the aimed inductive type (*e.g.* the type of *term*), Coq provides a number of destructors for *ident*. Destructors are named *ident*`_ind`, *ident*`_rec` or *ident*`_rect` which respectively correspond to elimination principles on `Prop`, `Set` and `Type`. Note that *ident*`_ind` is always provided whereas *ident*`_rec` and *ident*`_rect` are not. The type of the destructors expresses structural induction/recursion principles over objects of *ident*. The inductive definitions are formally detailed in section 6.5. We give below two examples of the use of the `Inductive` definitions.

The set of natural numbers is defined as :

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

The type `nat` is defined as the least `Set` containing `O` and closed by the `S` constructor. The constants `nat`, `O` and `S` are added to the environment.

Now let us have a look at the elimination principles. They are three : `nat_ind`, `nat_rec` and `nat_rect`. The type of `nat_ind` is :

```
Coq < Check nat_ind.
```

This is the well known structural induction principle over natural numbers, i.e. the second-order form of Peano's induction principle. It allows to prove some universal property of natural numbers (`(n:nat)(P n)`) by induction on `n`. Recall that `(n:nat)(P n)` is Gallina's syntax for the universal quantification $\forall n : nat \cdot P(n)$.

The types of `nat_rec` and `nat_rect` are similar, except that they pertain to `(P:nat->Set)` and `(P:nat->Type)` respectively . They correspond to primitive induction principles (allowing dependent types) respectively over sorts `Set` and `Type`.

As a second example, let us define the *even* predicate :

```
Coq < Inductive even : nat->Prop :=
Coq <     even_O  : (even O)
Coq <   | even_SS : (n:nat)(even n)->(even (S (S n))).
```

The type `nat->Prop` means that `even` is a unary predicate (inductively defined) over natural numbers. The type of its two constructors are the defining clauses of the predicate `even`. The type of `even_ind` is :

```
Coq < Check even_ind.
```

From a mathematical point of vue it asserts that the natural numbers satisfying the predicate `even` are just the naturals satisfying the clauses `even_0` or `even_SS`. This is why, when we want to prove any predicate `P` over elements of `even`, it is enough to prove it for `O` and to prove that if any natural number `n` satisfies `P` its double successor `(S (S n))` satisfies also `P`. This is indeed analogous to the structural induction principle we got for `nat`.

Error message :

1. `Non positive Occurrence in` $term_i$
2. `Type of Constructor not well-formed`

Variants :

1. `Inductive` *ident* `[` *params* `] :` *term* `:=` $ident_1$ `:` $term_1$ `| .. |` $ident_n$ `:` $term_n$ `.`
   Allows to define parameterized inductive types.
   The syntax of *params* is $ident'_1$`:`$term'_1$`, ... ,` $ident'_k$`:`$term'_k$.
   For instance, one can define parameterized lists as :

```
Coq < Inductive list [X:Set] : Set :=
Coq <  Nil : (list X) | Cons : X->(list X)->(list X).
```

   Note that, in the type of `Nil` and `Cons`, we write `(list X)` and not just `list`.
   The constants `Nil` and `Cons` will have respectively types

```
Coq < Check Nil.
```

   and

```
Coq < Check Cons.
```

   Types of destructors will be also quantified with `(X:Set)`.

2. `Inductive` *sort ident* `:=` $ident_1$ `:` $term_1$ `| .. |` $ident_n$ `:` $term_n$ `.`
   with *sort* being one of `Prop, Type, Set, Typeset` is equivalent to
   `Inductive` *ident* `:` *sort* `:=` $ident_1$ `:` $term_1$ `| .. |` $ident_n$ `:` $term_n$ `.`

3. `Inductive` *sort ident* `[` *params* `]:=` $ident_1$ `:` $term_1$ `| .. |` $ident_n$ `:` $term_n$ `.`
   Same as before but with parameters.

See also : sections 6.5, 4.6.1

2.5.2 Mutual Inductive

This command is a new feature of `Coq` V5.10. It allows to define mutually recursive inductive types. Its syntax is :

`Mutual Inductive` $ident_1$ `:` $term_1$ `:=`
  $ident^1_1$ `:` $term^1_1$
 `| ..`

| $ident^1_{n_1}$: $term^1_{n_1}$
with
..
with $ident_m$: $term_m$:=
$ident^m_1$: $term^m_1$
| ..
| $ident^m_{n_m}$: $term^m_{n_m}$.

It has the same semantics as the above `Inductive` definition for each $ident_1$, .., $ident_m$. All names $ident_1$, .., $ident_m$ are simultaneously added to the environment. Then well-typing of constructors can be checked. Each one of the $ident_1$, .., $ident_m$ can be used on its own.

It is also possible to parameterize these inductive definitions. However, one should remark that parameters correspond to a local context in which the whole set of inductive declarations is done. For this reason, the parameters are shared between all inductive types and this context syntactically appears between the `Mutual` and the `Inductive` keywords and not after the identifier as it is the case for a single inductive declaration. The syntax is thus:

`Mutual [`*params* `] Inductive` $ident_1$: $term_1$:=
$ident^1_1$: $term^1_1$
| ..
| $ident^1_{n_1}$: $term^1_{n_1}$
with
..
with $ident_m$: $term_m$:=
$ident^m_1$: $term^m_1$
| ..
| $ident^m_{n_m}$: $term^m_{n_m}$.

Example : The typical example of a mutual inductive data type is the one for trees and forests. We assume given two types A and B as variables. It can be declared the following way.

```
Coq < Variables A,B:Set.
```

```
Coq < Mutual Inductive tree : Set :=  node : A -> forest -> tree
Coq < with forest : Set := leaf : B -> forest
Coq <                    | cons : tree -> forest -> forest.
```

This declaration generates automatically six induction principles called respectively `tree_rec`, `tree_ind`, `tree_rect`, `forest_rec`, `forest_ind`, `forest_rect`. These ones are not the most general ones but are just the induction principles corresponding to each inductive part seen as a single inductive definition.

To illustrate this point on our example, we give the types of `tree_rec` and `forest_rec`.

```
Coq < Check tree_rec.
```

```
Coq < Check forest_rec.
```

Assume we want to parameterized our mutual inductive definitions with the two type variables A and B, the declaration should be done the following way:

```
Coq < Mutual [A,B:Set] Inductive
Coq <      tree : Set :=  node : A -> (forest A B) -> (tree A B)
Coq < with forest : Set := leaf : B -> (forest A B)
Coq <                    | cons : (tree A B) -> (forest A B) -> (forest A B).
```

Assume we define an inductive definition inside a section. When the section is closed, the variables declared in the section and occuring free in the declaration are added as parameters to the inductive definition.

2.5.3 The Record Macro

This version of Coq contains a macro called `Record` allowing the definition of records as is done in many programming languages. Its syntax is :

`Record` *ident* `[` *params* `]` `:` *sort* `:=` $ident_0$ `{`
  $ident_1$ `:` $term_1$`;`
  `...`
  $ident_n$ `:` $term_n$ `}.`

The identifier *ident* is the name of the defined record and *sort* is its type. The identifier $ident_0$ is the name of its constructor. The identifiers $ident_1$, .., $ident_n$ are the names of its fields and $term_1$, .., $term_n$ their respective types. Note that the records may have parameters.

Example :
The set of rational numbers is defined by :

```
Coq < Record Rat : Set := mkRat {
Coq <     top      : nat;
Coq <     bottom   : nat;
Coq <     Rat_cond : (gt bottom O) }.
```

An important difference between our records and those of most programming languages is that a field may depend on other fields appearing before it. For instance in the above example, the field `Rat_cond` depends on the field `bottom`. Thus the order of the fields is important.

Let us now see the work done by the `Record` macro. First the macro generates a one-constructor inductive definition of the following form :

`Inductive` *ident* `[` *params* `]` `:` *sort* `:=`
  $ident_0$ `:` `(`$ident_1$`:`$term_1$`)` `..` `(`$ident_n$`:`$term_n$`)(`*ident* *params*`).`

To build a object of type *ident*, one should provide to the constructor $ident_0$ with n terms filling the fields of the record.

Let us define the rational 1/2. Following our definition, a rational number is defined by two natural numbers and a proof that the second is strictly positive. Thus we must prove that 2 is strictly positive. Let us just assume it as axiom. Try to prove it using tactics (see the chapter 3).

```
Coq < Axiom two_is_positive : (gt (S(S O)) O).
```

We have now all the ingredients to define 1/2 (we call it `half`).

```
Coq < Definition half := (mkRat (S O) (S(S O)) two_is_positive).

Coq < Check half.
```

The macro generates also, when it is possible, the projection functions for destructuring a object of type *ident* into its constituent fields. We give the field names to these projection functions.

For our example, these functions are `top`, `bottom` and `Rat_cond`. Let us show their behavior on `half`.

```
Coq < Compute (top half).

Coq < Compute (bottom half).

Coq < Compute (Rat_cond half).
```

In the case where the definition of a projection function $ident_i$ is impossible, a warning is printed.

Warning :

1. `Warning:` $ident_i$ `cannot be defined.`
   This message is followed by an explanation of this impossibility.
   There may be three reasons :

   (a) The name $ident_i$ already exists in environment (see section 2.3.1).

   (b) The body of $ident_i$ uses a incorrect elimination for *ident* (see sections 2.5.4 and 6.5.5).

   (c) `The projections [` *idents* `] were not defined.`
   The body of $term_i$ uses the projections *idents* which are not defined for one of these three reasons listed here.

Error message :

1. `A record cannot be recursive`
   The record name *ident* appears in the type of its fields.

   During the definition of the one-constructor inductive definition, all the errors of inductive definitions, as described in section 2.5, may occur.

Variants :

1. `Record` *ident* `[` *params* `] :` *sort* `:= {`
   $ident_1$ `:` $term_1$`;`
   `...`
   $ident_n$ `:` $term_n$ `}.`

   One can omit the constructor name in which case the system will use the name `Build_`*ident*.

2.5.4 `Fixpoint` *ident* `[` $ident_1$ `:` $term_1$ `]` `:` $term_2$ `:=` $term_3$`.`

This command is a new feature of `Coq` V5.10. It may be used to define inductive objects using a fixed point construction instead of the `Match` recursion operator. The meaning of this declaration is to define *ident* a recursive function with one argument $ident_1$ of type $term_1$ such that $(ident\ ident_1)$ has type $term_2$ and is equivalent to the expression $term_3$. The type of the *ident* is consequentely $(ident_1 : term_1)term_2$ and the value is equivalent to $[ident_1 : term_1]term_3$. The argument $ident_1$ (of type $term_1$) is called the *recursive variable* of *ident*. Its type should be an inductive definition.

To be accepted, a `Fixpoint` definition has to satisfy some syntactical constraints on this recursive variable. They are needed to ensure that the `Fixpoint` definition always terminates. For instance, one can define the addition function as :

```
Coq < Fixpoint add [n:nat] : nat->nat := [m:nat]<nat>Case n of m
Coq < [p:nat](add p m) end.
```

The `Case` operator matches a value (here `n`) with the various constructors of its (inductive) type. The remaining arguments give the respective values to be returned, as functions of the parameters of the corresponding constructor. Thus here when `n` equals `O` we return `m`, and when `n` equals `(S p)` we return `(add p m)`. The `Case` operator is described in detail in section 6.5.5. The system recognizes that in the inductive call `(add p m)` the first argument actually decreases because it is a *pattern variable* coming from `Case n of`.

Variants :

- `Fixpoint` *ident* `[` *params* `]` `:` $term_1$ `:=` $term_2$`.`
  Assume that *params* is $ident'_1$ `:` $term'_1$`,` `..,` $ident'_k$ `:` $term'_k$`,` $ident'_0$ `:` $term'_0$. Then $ident'_1$`,` `..,` $ident'_k$ are parameters usable in the definition body $term_2$ and $ident_0$ is the recursion variable.

- `Fixpoint` $ident_1$ `[` $params_1$ `]` `:` $term_1$ `:=` $term'_1$
  `with`
  `..`
  `with` $ident_m$ `[` $params_m$ `]` `:` $term_m$ `:=` $term'_m$
  Allows to define simultaneously $ident_1$, .., $ident_m$.

Example : The following definition is not correct:

```
Coq < Fixpoint wrongplus [n:nat] : nat->nat
Coq <    := [m:nat]<nat>Case m of n [p:nat](wrongplus n p) end.
```

because the declared decreasing argument `n` actually does not decrease in the recursive call. The function computing the addition over the second argument should rather be written:

```
Coq < Fixpoint plus [n,m:nat] : nat := <nat>Case m of n [p:nat](plus n p) end.
```

The ordinary match operation on natural numbers can be mimicked the following way.

```
Coq < Fixpoint nat_match [C:Set;f0:C;fS:nat->C->C;n:nat] : C
Coq <     := <C>Case n of f0 [p:nat](fS p (nat_match C f0 fS p)) end.
```

The recursive call may not only be on direct subterms of the recursive variable `n` but also on a deeper subterm and we can directly write the function `mod2` which gives the remainder modulo 2 of a natural number.

```
Coq < Fixpoint mod2 [n:nat] : nat
Coq <    := <nat>Case n of O [p:nat]<nat>Case p of (S O) [q:nat](mod2 q) end end.
```

In order to keep the strong normalisation property, the fixed point reduction will only be performed when the argument in position of the recursive variable (whose type should be in an inductive definition) starts with a constructor.

The `Fixpoint` construction enjoys also the `with` extension to define functions over mutually defined inductive types or more generally any mutually recursive definitions.

Example : The size of trees and forests can be defined the following way:

```
Coq < Fixpoint tree_size [t:tree] : nat :=
Coq <     <nat>Case t of [a:A][f:forest](S (forest_size f)) end
Coq < with forest_size [f:forest] : nat :=
Coq <     <nat>Case f of [b:B](S O)
Coq <                    [t:tree][f':forest](plus (tree_size t) (forest_size f'))
Coq <          end.
```

A generic command `Scheme` is able to build automatically various mutual induction principles. It is described in section 8.5.

2.5.5 `Recursive Definition ...`

This command is a new feature of Coq V5.10. It is a high level tool for defining recursive functions. Its syntax follows the schema :

`Recursive Definition` *ident* `:` *term* `:=`
  $pattern_1^1$ `..` $pattern_n^1$ `=>` $term_1$
`| ..`
`|` $pattern_1^m$ `..` $pattern_n^m$ `=>` $term_m$.

It can be compared to functions declarations in ML languages (see, for instance [73]).
The `Recursive Definition` command uses a heuristic which tries to derive a term satisfying the specification. That is to say that Coq tries to find a term `T` such that for each $i \in [1..m]$,

$$(\mathtt{T}\ pattern_1^i\ \ldots pattern_n^i) =_{\beta\iota\delta} term_i.$$

If such a term can be inferred then *ident* is defined as being `T` and all corresponding equational theorems are provided. We refer the reader to section 15.4 for more details.

2.6 Section mechanism

The sectioning mechanism allows to organize a proof in structured sections. Then local declarations become available (see section 2.4).

2.6.1 `Section` *ident*

This command is used to open a section named *ident*.

Variants :

1. `Chapter` *ident*
   Same as `Section` *ident*

2.6.2 `End` *ident*

This command closes the section named *ident*. When a section is closed, all local declarations are discharged. This means that all global objects defined in the section are *closed* (in the sense of λ-calculus) with as many abstractions as there were local declarations in the section explicitely occuring in the term. A local object in the section is not exported and its value will be substituted in the other definitions.

Here is an example :

```
Coq < Section s1.

Coq < Variables x,y : nat.

Coq < Local y' := y.

Coq < Definition x' := (S x).

Coq < Print x'.

Coq < End s1.

Coq < Print x'.
```

Note the difference between the value of `x'` inside section `s1` and outside.

Error message :

1. `Section` *ident* `does not exist (or is already closed)`

2. `Section` *ident* `is not the innermost section`

Remark : Some commands such as `Hint` *ident* or `Syntactic Definition` which appear inside a section are cancelled when the section is closed.

Chapter 3

Proof handling

In Coq's proof editing mode all toplevel commands remain available and the user has access to specialized commands dealing with proof development pragmas documented in this section. He can also use some other specialized commands called *tactics*. They are the very tools allowing the user to deal with logical reasoning. They are documented in chapter 4.
When switching in editing proof mode, the prompt `Coq <` is changed into *ident* `<` where *ident* is the declared name of the theorem (or lemma, ...) one wants to prove.

At each stage of a proof development, one has a list of goals to prove. Initially, the list consists only in the theorem itself. After having applied some tactics, the list of goals contains the subgoals generated by the tactics. At each state of a proof development one has a number of available hypotheses we call the *local context* of the goal. Initially, the local context is empty. It is enriched by the use of certain tactics (see mainly section 4.2.2). Different local contexts may be associated to differents subgoals (see, for instance, section 4.6.1).

When a proof is achieved the message `Subtree proved!` is displayed. One can then store this proof as a defined constant in the environment. Because there exists a correspondence between proofs and terms of λ-calculus, known as the *Curry-Howard isomorphism* [41, 3, 39, 44], Coq stores proofs as terms of Cic. One calls those terms : *proof terms*.
Error message : When one attempts to use a proof editing command out of the proof editing mode, Coq raises the error message : `No focused proof`.

3.1 Switching on/off the proof editing mode

3.1.1 `Goal` *term*

This command switches Coq to editing proof mode and sets *term* as the original goal. It associates the name `Unnamed_thm` to the unnamed goal *term*.

Error message :

1. `Proof objects can only be abstracted`
2. `A goal should be a type`
3. `repeated goal not permitted in refining mode`

See also : section 3.1.3

3.1.2 `Qed`

This command is available in interactive editing proof mode when the proof is completed. Then `Qed` extracts a proof term from the proof script, switches back to `Coq` toplevel and attaches the extracted proof term to the declared name of the original goal. This name is added to the environment as an `Opaque` constant.

Error message :

1. `Attempt to save an incomplete proof`
2. `Clash with previous constant ...`
   The implicit name is already defined. You have then to provide explicitly a new name (see variant 2 below).

Variants :

1. `Save`
   Is equivalent to `Qed`.
2. `Save` *ident*
   Forces the name of the original goal to be *ident*.
3. `Save Theorem` *ident*
   Is equivalent to `Save` *ident*
4. `Save Remark` *ident*
   Defines the proved term as a local constant that will not exist anymore after the end of the current section.
5. `Defined`
   Defines the proved term as a transparent constant.

3.1.3 `Theorem` *ident* `:` *term*.

This command switches to interactive editing proof mode and declares *ident* as being the name of the original goal *term*. When declared as a `Theorem`, the name *ident* is known at all section levels: `Theorem` is a *global* lemma.

Error message : (see section 3.1.1)
Variants :

1. `Lemma` *ident* `:` *term*
   Is equivalent to `Theorem` *ident* `:` *term*
2. `Remark` *ident* `:` *term*
   Analogous to `Theorem` except that *ident* will be unknown after closing the current section.
3. `Fact` *ident* `:` *term*
   Analogous to `Theorem` except that *ident* will be unknown after closing the section which is above the current section but known after closing the current section.

4. `Definition` *ident* `:` *term*
   Analogous to `Theorem`, intended to be used in conjunction with `Defined` (see chapter 5 in order to define a transparent constant).

3.2 Pragmas

3.2.1 `Proof` *term*

This command applies in proof editing mode. It is equivalent to `Exact` *term*`; Save.` That is, you have to give the full proof in one gulp, as a proof term (see section 4.1.1).

3.2.2 `Abort`

This command cancels the current proof development, switching back to the previous proof development, or to the Coq toplevel if no other proof was edited.

Error message :

1. `No focused proof (No proof-editing in progress)`

Variants :

1. `Abort` *ident*
   Aborts the editing of the proof named *ident*.

2. `Abort All`
   Aborts all current goals, switching back to the Coq toplevel.

3.2.3 `Suspend`

This command applies in proof editing mode. It switches back to the Coq toplevel, but without cancelling the current proofs.

3.2.4 `Resume`

This commands switches back to the editing of the last edited proof.

Error message :

1. `No proof-editing in progress`

Variants :

1. `Resume` *ident*
   Restarts the editing of the proof named *ident*.

Error message :

1. `No such proof`

3.2.5 `Undo`

This command cancels the effect of the last tactic command. Thus, it backtracks one step.

Error message :

1. `No focused proof (No proof-editing in progress)`
2. `Undo stack would be exhausted`

Variants :

1. `Undo` *num*
   Repeats `Undo` *num* times.

3.2.6 `Restart`

This command restores the proof editing process to the original goal.

Error message :

1. `No focused proof to restart`

3.2.7 `Focus`

Will focus the attention on the first subgoal to prove, the remaining subgoals will no more be printed after the application of a tactic. This is useful when there are many current subgoals which clutter your screen.

3.2.8 `Unfocus`

Turn off the focus mode.

3.2.9 `Show`

This command displays the current goals.

Variants :

1. `Show` *num*
   Displays only the *num*-th subgoal.
   Error message : `no such goal`

3.2.10 `Clear` *ident*

This command erases the hypothesis named *ident* in the local context of the current goal. Then *ident* is no more displayed and no more usable in the proof development.

Error message :

1. *ident* `is not among the assumptions.`

3.3 The hints list

The hints list is a data base of tactics for automated proof search. It associates to a constant a list of tactics which may be tried when the head symbol of the goal to be solved is this constant.

The tactics that can be stored are mainly `Apply` *ident* (see section 4.3.3), `Exact` *ident* (see section 4.1.1), or `Unfold` *ident* (see section 4.4.4).
Each tactic is stored with a numerical weight aiming to represent the "cost" of the application of this tactic in an automatic proof search. Tactics with a low cost are tried first.
See also : section 4.9

3.3.1 `Hint` *ident*

This command adds `Apply` *ident* to the hint list associated with the head symbol of the type of *ident*. The cost of *ident* is the number of subgoals generated by `Apply` *ident*.

In case the inferred type of *ident* does not start with a product the tactic added in the hint list is `Exact` *ident*. In case this type can be reduced to a type starting with a product, the tactic `Apply` *ident* is also stored in the hints list.

Error message :

1. `Bound head variable`
   The head symbol of the type of *ident* is a bound variable such that this tactic cannot be associated to a constant.

2. *ident* `cannot be used as a hint`
   The type of *ident* contains products over variables which do not appear in the conclusion. A typical example is a transitivity axiom. In that case the `Apply` tactic fails, and thus is useless.

Variants :

1. `Hint` $ident_1$ `..` $ident_n$
   Is equivalent to `Hint` $ident_1$`.` `..` `Hint` $ident_n$

3.3.2 `Immediate` *ident*

This command adds `Apply` *ident*`;` `Trivial` to the hint list associated with the head symbol of the type of *ident*. This tactic will fail if all the subgoals generated by `Apply` *ident* are not solved immediately by the `Trivial` tactic which only tries priority 0 tactics.

This command is useful for theorems such that the symmetry of equality or $n+1=m+1 \rightarrow n=m$ that we may like to introduce with a limited use in order to avoid useless proof-search.

The cost of this tactic (which never generates subgoals) is always 1, so that it is not used by `Trivial` itself.

Error message :

1. `Bound head variable`

Variants :

1. `Immediate` $ident_1$ `..` $ident_n$
   Is equivalent to `Immediate` $ident_1$`.` `..` `Immediate` $ident_n$

3.3.3 `Hint Unfold` *ident*

This command adds the tactic **`Unfold`** *ident* to the hint list that will only be used when the head constant of the goal is *ident*. Its cost is 4.
Variants :

1. **`Hint Unfold`** $ident_1$ `..` $ident_n$
   Is equivalent to **`Hint Unfold`** $ident_1$**`.`** `..` **`Hint Unfold`** $ident_n$

3.3.4 `Print Hint`

This command displays the currently available hints list. Note that if an axiom or theorem has been declared twice, it will appear only once.

Chapter 4

Tactics

A deduction rule is a link between some (unique) formula, we call the *conclusion* and (several) formulæ we call the *premisses*. Indeed, a deduction rule can be read in two ways. The first one has the shape : *"if I know this and this then I can deduce this"*. For instance, if I have a proof of A and a proof of B then I have a proof of $A \wedge B$. This is forward reasoning from premisses to conclusion. The other way says : *"to prove this I have to prove that and that"*. For instance, to prove $A \wedge B$, I have to prove A and I have to prove B. This is backward reasoning which proceeds from conclusion to premisses. We say that the conclusion is *the goal* to prove and premisses are *the subgoals*. The tactics implement *backward reasoning*. When applied to a goal, a tactic replaces this goal with the subgoals it generates. We say that a tactic reduces a goal to its subgoal(s).

Each (sub)goal is denoted with a number. The current goal is numbered 1. By default, a tactic is applied to the current goal, but one can address a particular goal in the list by writing *n:tac* which means *"apply tactic tac to goal number n"*.

Since not every rule applies to any statement, every tactic cannot be used to reduce any goal. In other words, before applying a tactic to a given goal, the system checks that some *preconditions* are satisfied. If it is not the case, the tactic raises an error message.

There are, at least, three levels of tactics. The simplest one implements basic rules of the logical framework (see for instance `Intro` in section 4.2.2). The second level is the one of *derived rules* which are built by combination of other tactics (see for instance `Generalize` in section 4.3.1). The third one implements heuristics or decision procedures to build a complete proof of a goal (see for instance `Auto` in section 4.9.1).

4.1 Brute force proofs

4.1.1 `Exact` *term*.

This tactic applies to any goal. It gives directly the exact proof term of the goal. Let `T` be our goal, let `p` be a term of type `U` then `Exact p` succeeds iff `T` and `U` are convertible.

Error message :

1. `Not an exact proof`

4.2 Basics

Tactics presented in this section implement the basic typing rules of CIC given in chapter 6.

4.2.1 `Assumption.`

This tactic applies to any goal. It implements the "Var" rule given in section 6.2. It looks in the local context for an hypothesis which type is equal to the goal. If it is the case, the proof is ended and the message `Subtree proved!` is displayed.

Error message :

1. `No such assumption`

4.2.2 `Intro.`

This tactic applies to a goal which is a product. It implements the "Lam" rule given in section 6.2. In fact, only one subgoal will be generated as the other one can be automatically checked.

If the current goal is a dependent product (say : `(x:T)U`) and `x` is a name that does not exist in the current context, then `Intro` puts `x:T` in the local context. Otherwise, it puts `x`*n*`:T` where `x`*n* is a fresh name.

If the goal is a non dependent product (say : `T -> U`) then it puts in the local context either `H`*n*`:T` (if the type of `T` is `Set` or `Prop`) or `X`*n*`:T` (if the type of `T` is `Typeset` or `Type`) where `H`*n* and `X`*n* are fresh identifiers.

In both cases the new subgoal is `U`.

Remark : In the case you have a non dependent product as a goal but you entered it under the form of a dependent one (say: your entered `(x:T)U` where `x` does not occur in `U`) you will see the goal printed as `T->U` but `Intro` will work as in the dependent case.

Error message :

1. `Intro needs a product`

Variants :

1. `Intros.`
   Repeats `Intro` as often as it is possible. It is equivalent to the tactical `Repeat Intro`.

2. `Intro` *ident*`.`
   Applies `Intro` but forces *ident* to be the name of the hypothesis.
   Error message : `name` *ident* `is already bound`
   Remark : `Intro` doesn't check the whole current context. Actually, identifiers declared or defined in required modules can be used as *ident* and, in this case, the old *ident* of the module is no more reachable.

3. `Intros` $ident_1$ `..` $ident_n$`.`
   Is equivalent to the tactical `Intro` $ident_1$`; .. ; Intro` $ident_n$.

4. `Intros until` *ident*.
   Repeats `Intro` until it meets a premiss of the goal having form (*ident* : *term*) discharges the variable named *ident* of the current goal.
   Error message : `No such hypothesis in current goal`

4.2.3 `Cut` *term*.

This tactic applies to any goal. It implements the "App" rule given in section 6.2. It is used when one wants to prove the current goal (say : `T`) as a consequence of a statement `U`. That is to say that `Cut U` transforms the current goal `T` into the two following subgoals : `U -> T` and `U`.

Error message :

1. `Not a proposition or a type`
   Arises when the argument *term* is neither of type `Prop`, `Set`, `Type` nor `Typeset`.

4.2.4 `Change` *term*.

This tactic applies to any goal. It implements the rule "Conv" given in section 6.3. `Change U` replaces the current goal (say : `T`) with a `U` providing that `U` is well-formed and that `T` and `U` are convertible.

Error message :

1. `convert-concl rule passed non-converting term`

Variants :

1. `Change` *term* `in` *ident*.
   Not yet installed.

4.3 Some derived rules

4.3.1 `Generalize` *term*.

This tactic applies to any goal. Its main use is to enforce the current goal with a quantification. In this case, *term* must be the name of a variable of the local context on which depends the current goal. Assume that our current goal is some `(P x)` and that the local context contains `x:T`, then `Generalize x` transforms the current goal into `(x:T)(P x)`.
Remark : If *term* is not the name of a variable of the local context then `Generalize t` is equivalent to the tactical `Cut T; 2: Exact t` where `T` is the type of `t`.

Variants :

1. `Generalize` $ident_1$ `..` $ident_n$.
   Is equivalent to `Generalize` $ident_n$`; .. ; Generalize` $ident_1$. Note that the $ident_i$'s are processed from n to 1.

4.3.2 `Specialize` *term*.

Is equivalent to `Generalize` *term*.

Variants :

1. `Specialize` *term* `with` ref_1 `:=` $term_1$ `..` ref_n `:=` $term_n$.
   It is to provide the tactic with some explicit values to instantiate premisses of *term* (see section 4.3.6).
   Error message : `Metavariable wasn't in the metamap`
   Arises when the informations provided in the binding list is not enough.

2. `Specialize` *num term* `with` *ref*`:=` *term* `..` *ref*`:=` *term*.
   No yet documented.

4.3.3 `Apply` *term*.

This tactic applies to any goal. The argument *term* can be either an hypothesis of the proof context or a constant of the environment (axiom, theorem, ..). The tactic `Apply` tries to match the current goal against the conclusion of the type *term*. If it succeeds, then the tactic returns as many subgoals as the instantiations of the premisses of the type of *term* which are not simply hypotheses from the proof context.

Error message :

1. `Impossible to unify ... with ...`
   Since higher order unification is undecidable, the `Apply` tactic may fail when you think it should work. In this case, if you know that the conclusion of *term* and the current goal are unifiable, you can help the `Apply` tactic by transforming your goal with the `Change` or `Pattern` tactics (see sections 4.4.5, 4.2.4).

2. `Cannot refine to conclusions with meta-variables`
   This occurs when some instantiations of premisses of *term* are not deducible from the unification. This is the case, for instance, when you want to apply a transitivity property. In this case, you have to use the variant below : `Apply .. with`.

Variants :

1. `Apply` *term* `with` $term_1$ `..` $term_n$.
   Provides `Apply` with explicit instantiations for all dependent premisses of the type of *term* which do not occur in the conclusion and consequently cannot be found by unification. Notice that $term_1$ .. $term_n$ must be given according to the order of premisses of the type of *term*.
   Error message : `Not the right number of missing arguments`

2. `Apply` *term* `with` ref_1 `:=` $term_1$ `..` ref_n `:=` $term_n$.
   Provides also `Apply` with values for instantiating premisses by associating explicitly variables (or non dependent products) with their intended instance (see syntax in the next section).

4.3.4 `Absurd` *term*.

This tactic applies to any goal. The argument *term* is any proposition `P` of type `Prop`. This tactic applies `False` elimination, that is it deduces `P` from `False`, assuming that the current context is inconsistent. It generates as subgoals ~`P` and `P`. It is very useful in proofs by cases, where certain cases are impossible. Typically, when an hypothesis `H` is such that ~`H` may be deduced from the rest of the context, `Absurd H; Assumption` will leave you with this sole proof obligation, independently of the current goal.

Remark : It could be generalized to `Set` by the use of the axiom `False_rec`.

4.3.5 `Contradiction`.

This tactic applies to any goal. The `Contradiction` tactic attempts to find in the current context (after all `Intros`) one which is equivalent to `False`. It permits to prune irrelevant cases. This tactic is a macro for the tactics sequence `Intros; ElimType False; Assumption`.

Error message :

1. `No such assumption` : when there is no assumption in the context that is equivalent to `False`.

4.3.6 Binding list

A binding list is generally used after the `with` keyword in tactics. The general shape of a binding list is ref_1 `:=` $term_1$ `..` ref_n `:=` $term_n$ where *ref* is either an *ident* or a *num*. It is used to provide a tactic with a list of values ($term_1$, .., $term_n$) that have to be substituted respectively to ref_1, .., ref_n. For all $i \in [1..n]$, if ref_i is $ident_i$ then it references the dependent product $ident_i$`:T` (for some type `T`); if ref_i is num_i then it references the num_ith non dependent premiss.

4.4 Conversion tactics

This set of tactics implements different restricted usages of the “Conv” rule given in section 6.3.

4.4.1 `Red`.

This tactic applies to a goal which have form `(x:T1)..(xk:Tk)(c t1 .. tn)` where `c` is a constant. If `c` is transparent then it replaces `c` with its definition (say `t`) and then reduces `(t t1 .. tn)` according to $\beta\iota$-reduction rules.

Error message :

1. `Term not reducible`

Variants :

1. `Red in` *ident*.
   Applies `Red` to the hypothesis named *ident*.

4.4.2 `Hnf.`

This tactic applies to any goal. It replaces the current goal with its head normal form according to the $\beta\delta\iota$-reduction rules. `Hnf` does not produce a real head normal form but either a product or an applicative term in head normal form or a variable.
Example : The term `(n:nat)(plus (S n) (S n))` is not reduced by `Hnf`.
Remark : The δ rule will only be applied to transparent constants (i.e. which have not been frozen with an `Opaque` command; see section 5.6.1).

4.4.3 `Simpl.`

This tactic applies to any goal. Let `T` be our current goal. The tactic `Simpl` first applies $\beta\iota$-reduction rule to transform `T` into, say, `T'`. Then it expands transparent constants and tries to reduce `T'` according, once more, to $\beta\iota$ rules. But when the ι rule is not applicable then possible δ-reductions are not applied. For instance trying to use `Simpl` on `(plus n O)=n` will change nothing.

Variants :

1. `Simpl in` *ident*`.`
   Applies `Simpl` to the hypothesis named *ident*.

4.4.4 `Unfold` *ident*`.`

This tactic applies to any goal. The argument *ident* must be the name of a defined transparent constant (see section 2.4). The tactic `Unfold` applies the δ rule to each occurrence of *ident* in the current goal and then replaces it with its $\beta\iota$-normal form.

Error message :

1. `Constant is opaque`

2. *ident* `does not occur`

Variants :

1. `Unfold` $ident_1$ `..` $ident_n$`.`
   Replaces *simultaneously* $ident_1$, .., $ident_n$ with their definitions and replaces the current goal with its $\beta\iota$ normal form.

2. `Unfold` num_1^1 `..` num_i^1 $ident_1$ `..` num_1^n `..` num_j^n $ident_n$`.`
   The lists num_1^1, .., num_i^1 and num_1^n, .., num_j^n are to specify the occurrences of $ident_1$, .., $ident_n$ to be unfold. Occurrences are located from left to right in the linear notation of terms.
   Error message : `bad occurrence numbers of` $ident_i$

4.4.5 `Pattern` *term*`.`

This command applies to any goal. The argument *term* must be a free subterm of the current goal. The command `Pattern` performs β-expansion of the current goal (say `T`) by

1. replacing all occurrences of *term* in `T` with a fresh variable

2. abstracting this variable

3. applying *term* to the abstracted goal

For instance, if `T` is `(P t)` when `t` does not occur in P then `Pattern t` transforms it into `([x:A](P x) t)`. This command has to be used, for instance, when an `Apply` command fails on matching.

Variants :

1. `Pattern` num_1 `..` num_n *term*`.`
   Only the occurrences num_1 .. num_n of *term* will be considered for β-expansion. Occurrences are located from left to right.

2. `Pattern` num^1_1 `..` num^1_i $term_1$ `..` num^m_1 `..` num^m_j $term_m$`.`
   Will process occurrences num^1_1, .., num^1_i of $term_1$, .., num^m_1, .., num^m_j of $term_m$ starting from $term_m$. Starting from a goal `(P` $\mathtt{t}_1 \ldots \mathtt{t}_m$`)` with the $\mathtt{t}_i$ which do not occur in P, the tactic `Pattern` $\mathtt{t}_1 \ldots \mathtt{t}_m$ generates the equivalent goal `([`$\mathtt{x}_1$`:`$\mathtt{A}_1$`]`...`[`$\mathtt{x}_m$`:`$\mathtt{A}_m$`](P` $\mathtt{x}_1 \ldots \mathtt{x}_m$`)` $\mathtt{t}_1 \ldots \mathtt{t}_m$`)`.
   If t_i occurs in one of the generated types A_j these occurrences will also be considered and possibly abstracted.

4.5 Introductions

Introduction tactics address goals which are inductive constants. They are used when one guesses that the goal can be obtained with one of its constructors' type.

4.5.1 `Constructor` *num*`.`

This tactic is a new feature of Coq V5.10. It applies to a goal such that the head of its conclusion is an inductive constant (say `I`). The argument *num* must be less or equal to the numbers of constructor(s) of `I`. Let `ci` be the `i`-th constructor of `I`, then `Constructor i` is equivalent to `Intros; Apply ci`.

Error message :

1. `Not an inductive product`

2. `Not enough Constructors`

Variants :

1. `Split.`
   Applies if `I` has only one constructor, typically in the case of conjunction $A \wedge B$. It is equivalent to `Constructor 1`.

2. `Left.`, `Right.`
   Apply if `I` has two constructors, for instance in the case of disjunction $A \vee B$. They are respectively equivalent to `Constructor 1` and `Constructor 2`.

3. `Exists` *term*`.`
   Applies if `I` has only one constructor, for instance in the case of existential quantification $\exists x \cdot P(x)$. Calling `c` this unique constructor, `Exists t` is equivalent to `Apply c with t`.

4.6 Eliminations

Elimination tactics are useful to prove statements by induction. Indeed, they make use of the elimination (or induction) principles generated with inductive definitions (see section 6.5).

4.6.1 `Elim` *term*.

This tactic applies to any goal. Basically, the type of the argument *term* must be an inductive constant. Then according to the type of the goal, the tactic `Elim` chooses the right destructor and applies it (as in the case of the `Apply` tactic). For instance, assume that our proof context contains `n:nat`, assume that our current goal is `T`, with `T` of type `nat->Prop`, then `Elim n` is equivalent to `Apply nat_ind`.

Error message : :

1. `Not an inductive product`

2. `Cannot refine to conclusions with meta-variables`
   As `Elim` uses `Apply`, see section 4.3.3 and the variant `Elim .. with ..` below.

Variants :

1. `Elim` *term* also works when the type of *term* starts with a product and the head symbol is an inductive definition. In that case the tactic tries both to find an object in the inductive definition and to use this inductive definition for elimination. In case of non-dependent products in the type, subgoals are generated corresponding to the hypotheses. In the case of dependent products, the tactic will try to find an instance for which the elimination lemma applies.

2. `Elim` *term* `with` $term_1$ `..` $term_n$.
   Allows the user to give explicitly the values for dependent premisses of the elimination schema. All arguments must be given.
   Error message : `Not the right number of dependent arguments`

3. `ElimType` *term*.
   The argument *term* must be inductively defined. `ElimType I` is equivalent to `Cut I; Intro H`*n*`; Elim H`*n*. But the hypothesis `H`*n* will not appear in the context(s) of the subgoal(s).
   Conversely, if `t` is a term of (inductive) type `I` then `Elim t` is equivalent to `ElimType I; 2: Exact t.`
   Error message : `simpl Impossible to unify ... with ...` Arises when *term* needs to be applied to parameters.

4. `Induction` *ident*.
   Is equivalent to `Intros until` *ident*; `Pattern` *ident*; `Elim` *ident*.

5. `Induction` *num*.
   Is analogous to `Induction` *ident* but for the *num*-th non-dependent premiss of the goal.

4.6.2 `Case` *term*.

This tactic is a new feature of Coq V5.10. The type of *term* must be inductively defined. The tactic `Case` is used to perform case analysis without recursion.

Variants :

1. `Case` *term* `with` $term_1$ `..` $term_n$.
   Analogous to `Elim .. with` above.

2. `Destruct` *ident*.
   Is equivalent to the tactical `Intro Until` *ident*; `Case` *ident*.

3. `Destruct` *num*.
   Is equivalent to `Destruct` *ident* but for the *num*-th non dependent premiss of the goal.

4.7 Equality

These tactics use the predefined equalities `eq:(A:Set)A->A->Prop` defined in file `Logic.v` (see section refEquality), and implicitly used in the syntax `t=u` and `eqT : (A:Type)A->A->Prop` defined in file `Logic_Type.v`, and implicitly used in the syntax `t==u`. In the following, the notation `t=u` will represent either one of these two equalities.

4.7.1 `Rewrite` *term*.

This tactic applies to any goal. The conclusion of the type of *term* must have the conclusion $term_1$=$term_2$. Then `Rewrite` *term* replaces every occurrence of $term_1$ by $term_2$ in the goal.
Remark : In case $term_2$ contains occurrences of variables bound in the type of *term*, the tactic tries first to find a subterm of the goal which matches this term in order to find a closed instance $term'_2$ of $term_2$ then all instances of $term'_2$ will be replaced.

Error message :

1. `No equality here`

Variants :

1. `Rewrite ->` *term*.
   Is equivalent to `Rewrite` *term*

2. `Rewrite <-` *term*.
   Uses the equality $term_1$=$term_2$ from right to left

4.7.2 `Replace` $term_1$ `with` $term_2$.

This tactic applies to any goal. It replaces all free occurrences of $term_1$ in the current goal with $term_2$ and generates the equality $term_2$=$term_1$ as a subgoal. It is equivalent to `Cut` $term_1$=$term_2$; `Intro H`*n*; `Rewrite H`*n*. `Clear H`*n*.

4.7.3 `Reflexivity.`

This tactic applies to a goal which has the form `t=u`. It checks that `t` and `u` are convertible. It is equivalent to `Apply refl_equal`.

Error message :

1. `Not a predefined equality`
2. `Impossible to unify ... With ...`

4.7.4 `Symmetry.`

This tactic applies to a goal which have form `t=u` and changes it into `u=t`.

4.7.5 `Transitivity` *term*.

This tactic applies to a goal which have form `t=u` and transforms it into the two subgoals `t=`*term* and *term*`=u`.

4.8 Equality and inductive sets

We describe in this section some special purpose tactics dealing with equality and inductive sets.

4.8.1 `Simple Discriminate.`

This tactic applies to a goal which has the form $\sim term_1 = term_2$. The terms $term_1$ and $term_2$ must belong to an inductive set. The `Simple Discriminate` tactic is a special purpose tactic for proving trivial disequalities such as `~O=(S n)`. It checks that the head symbols of the head normal forms of $term_1$ and $term_2$ are not the same constructor. When this is the case, the current goal is solved.

Error message :

1. `Simple Discriminate should be applied to a pair of terms built with different constructors`

4.8.2 `Discriminate` *ident*

This is a special purpose tactic for proving any goal from an absurd hypothesis stating that, two structurally different terms of an inductive set are equal. For example, from the hypothesis `(S (S O))=(S O)` we can derive by absurdity any proposition. Let *ident* be a hypothesis of type $term_1$ `=` $term_2$ in the local context, $term_1$ and $term_2$ are elements of an inductive set. To build the proof, the tactic traverses the normal forms* of $term_1$ and $term_2$ looking for a couple of subterms `u` and `w` (`u` subterm of the normal form of $term_1$ and `w` subterm of the normal form of $term_2$), placed respectively in the same positions and, whose head symbols are different constructors. If such a couple of subterms exists, then the proof of the current goal is ended and the message `Subtree proved!` is displayed, otherwise the tactic fails arising an error message.

*Recall: opaque constants will not be expanded by δ reductions

Error message :

1. *id* `Not a discriminable equality` occurs when the type of the specified hypothesis is an equation but does not verify the expected preconditions.

2. *id* `Not an equation` occurs when the type of the specified hypothesis is not an equation.

4.8.3 `Discriminate.`

`Discriminate` applies to a goal of the form $\sim$ *term*$_1$=*term*$_2$ and its semantics is equivalent to the sequence : `Unfold not; Intro` *ident* ; `Discriminate` *ident*.

Error message :

1. `goal does not satisfy the expected preconditions.`

4.8.4 `Injection` *ident* *num*$_1$ `..` *num*$_n$`.`

This tactic applies to a goal which has the form *term*$_1$ `=` *term*$_2$. The terms *term*$_1$ and *term*$_2$ must belong to an inductive type. The name *ident* must be the name of an hypothesis whose type is some `t=u`. The sequence *num*$_1$ .. *num*$_n$ is *a position* (or a *path*) in `t` and `u`. The `Injection` tactic is based on the fact that constructors of inductive sets are injections. That means that if c is a constructor of an inductive set, and $(c\ \vec{t_1})$ and $(c\ \vec{t_2})$ are two terms that are equal then $\vec{t_1}$ and $\vec{t_2}$ are equal too.
Then, the tactic `Injection` checks that the normal forms of *term*$_1$ and *term*$_2$ are subterms of position *num*$_1$ .. *num*$_n$ in the normal form of (respectively) `t` and `u`, then it checks that the path from the roots of `t` and `u` to (respectively) *term*$_1$ and *term*$_2$, always meets the same constructor.
For instance, under the hypothesis `H:(S n)=(S m)`, the tactic `Injection H 1` will prove `n=m`.

Error message :

1. `not an equality`

2. `incorrect path`
   Arises when the given path *num*$_1$ .. *num*$_n$ exceeds the depth of the current goal

3. `can not perform injection in the specified hypothesis`
   Arises when the specified hypothesis does not have the expected type

4. `the result of the injection does not correspond to the current subgoal`
   Arises when the equality resulting from the injection is not convertible with the current goal

4.8.5 `Injection` *ident*

If *ident* is an hypothesis of type *term*$_1$ `=` *term*$_2$, then this tactic behaves as applying injection as deep as possible to derive the equality of all the subterms of *term*$_1$ and *term*$_2$ placed in the same positions. For example, from the hypothesis `(S (S n))=(S (S (S m))` we may derive `n=(S m)`. To use this tactic *term*$_1$ and *term*$_2$ should be elements of an inductive set and they should be neither explicitly equal, nor structurally different. We mean by this that, if $\mathtt{n}_1$ and $\mathtt{n}_2$ are their respective normal forms, then :

- n_1 and n_2 should not be syntactically equal,
- there must not exist any couple of subterms `u` and `w`, `u` subterm of n_1 and `w` subterm of n_2 , placed in the same positions and having different constructors as head symbols.

If these conditions are satisfied, then, the tactic derives the equality of all the subterms of $term_1$ and $term_2$ placed in the same positions and puts them as antecedents of the current goal. Beware that `Injection` yields always an equality in a sigma type whenever the injected object has a dependent type.

Example : Let's consider the type of dependent lists :

```
Coq < Inductive listn : nat->Set  :=
Coq <          nil: (listn O) | cons: (n:nat) nat-> (listn n)-> (listn (S n)).
```

`Injection H` derives that `n=O`, but instead of the equality `l=nil` it yields an equality in the sigma type `{n:nat & (listn n)}` :

```
(existS nat [n0:nat](listn n0) O nil)=(existS nat [n0:nat](listn n0) O l)
```

By now, we do not have any tactic that given the hypothesis `(cons O n nil)=(cons O O l)` can derive the equality in the dependent type `<(listn O)>nil=l`. These kind of equalities can be used to perform rewriting. †

Error message :

1. *id* `is not a projectable equality` occurs when the type of the hypothesis *id* does not verify the preconditions.
2. *id* `Not an equation` occurs when the type of the hypothesis *id* is not an equation.

4.8.6 `Injection.`

If the current goal is of the form $\sim$ $term_1$=$term_2$, the tactic calculates the head normal form of the goal and then behaves as the sequence: `Unfold not; Intro` *ident*; `Injection` *ident*.

Error message : `goal does not satisfy the expected preconditions`

4.8.7 `Simplify_eq` *ident*

Let *ident* be the name of a hypothesis of type $term_1$=$term_2$ in the local context. If $term_1$ and $term_2$ are structurally different (in the sense described for the tactic `Discriminate`), then, `Simplify_eq` behaves as `Discriminate` *ident* otherwise it behaves as `Injection` *ident*.

4.8.8 `Simplify_eq.`

This tactic is defined on top of the previous one. If the current goal is of the form $\sim$ $t_1 = t_2$, then this tactic calculates the head normal form of the goal (like with the tactic `Hnf`) and then behaves as the sequence `Intro` *ident*; `Simplify_eq` *ident*.

† In the file `equality.mli` you can find the functions `Subst`, `HypSubst`, `RevSubst` and `RevHypSubst` that perform this kind of rewriting.

4.9 Automatizing

4.9.1 `Auto.`

This tactic implements a Prolog-like resolution procedure to solve the current goal. It first tries to solve the goal using the `Assumption` tactic, then it reduces the goal to an atomic one using `Intros` and introducing the newly generated hypotheses as hints. Then it looks at the list of tactics associated to the head symbol of the goal and tries to apply one of them (starting from the tactics with lower cost). This process is recursively applied to the generated subgoals. The maximal search depth is 5 by default.

Variants :

1. `Auto` *num*
   Forces the search depth to be *num*.

Remark : `Auto` either solves the goal or else acts as `Idtac` and does not change the goal.
See also : section 3.3

4.9.2 `Trivial.`

This tactic is a restriction of `Auto` for doing hypotheses and hints of cost 0. Typically it solves goals such as trivial equalities $X = X$.
See also : section 3.3

4.9.3 `Prolog [` $term_1 \ldots term_n$ `]` *num*.

This tactic, implemented by Chet Murthy, is based upon the concept of existential variables of Gilles Dowek, stating that resolution is a kind of unification. It tries to solve the current goal using the `Assumption` tactic, the `Intro` tactic, and applying hypotheses of the local context and terms of the given list `[` $term_1 \ldots term_n$ `]`. It is more powerful than `Auto` since it may apply to any theorem, even those of the form `(x:A)(P x) -> Q` where `x` does not appear free in `Q`. The maximal search depth is *num*.

Error message :

1. `Prolog failed`
   The Prolog tactic was not able to prove the subgoal.

4.9.4 `Tauto.`

This tactic, due to César Muñoz [57], implements a decision procedure for intuitionistic propositional calculus based on the contraction-free sequent calculi LJT* of R. Dyckhoff [33]. Note that `Tauto` succeeds on any instance of an intuitionistic tautological proposition such as `(x:nat)x=O -> x=O`.

4.9.5 Intuition.

The tactic Intuition takes advantage of the search-tree builded by the decision procedure involved in the tactic Tauto. It uses this information to generate a set of subgoals equivalent to the original one (but simpler than it) and applies the tactic Auto to them [57].

For instance, the tactic Intuition applied to the goal

```
((x:nat)(P x))/\B->((y:nat)(P y))/\(P O)\/B/\(P O)
```

replaces it by the equivalent one:

```
((x:nat)(P x) -> B -> (P O))
```

and then uses Auto which completes the proof.

4.9.6 Linear.

The tactic Linear, due to Jean-Christophe Filliâtre [34], implements a decision procedure for *Direct Predicate Calculus*, that is first-order Gentzen's Sequent Calculus without contraction rules [49, 8]. Intuitively, a first-order goal is provable in Direct Predicate Calculus if it can be proved using each hypothesis at most once.

Unlike the previous tactics, the Linear tactic does not belong to the initial state of the system, and it must be loaded explicitly with the command

```
Coq < Cd "$COQTOP/tactics/contrib/linear".
```

```
Coq < Require Linear.
```

For instance, assuming that even and odd are two predicates on natural numbers, and a of type nat, the tactic Linear solves the following goal

```
Coq < Lemma example : (even a)
Coq <                 -> ((x:nat)((even x)->(odd (S x))))
Coq <                 -> (Ex [y:nat](odd y)).
```

You can find examples of the use of Linear in theories/DEMOS/DemoLinear.v.

Variants :

1. Linear with *ident*$_1$ .. *ident*$_n$.
   Is equivalent to apply first Generalize *ident*$_1$ .. *ident*$_n$ (see section 4.3.1) then the Linear tactic. So one can use axioms, lemmas or hypotheses of the local context with Linear in this way.

Error message :

1. Not provable in Direct Predicate Calculus

2. Found n classical proof(s) but no intuitionnistic one !
   The decision procedure looks actually for classical proofs of the goals, and then checks that they are intuitionnistic. In that case, classical proofs have been found, which do not correspond to intuitionnistic ones.

4.10 Developing certified program

This section is devoted to powerful tools that `Coq` provides to develop certified programs. We just mention below the main features of those tools and refer the reader to chapter 12 and references [60, 61] for more details and examples.

4.10.1 `Realizer` *Fwterm*.

This command associates the term *Fwterm* to the current goal. The *Fwterm*'s syntax is described in the chapter 12. It is an extension of the basic syntax for `Coq`'s terms. The `Realizer` is used as a hint by the `Program` tactic described below. The term *Fwterm* intends to be the program extracted from the proof we want to develop.
See also : chapter 12, section 5.6.6

4.10.2 `Program.`

This tactic tries to make a one step inference according to the structure of the `Realizer` associated to the current goal.

Variants :

1. `Program_all.`
   Is equivalent to `Repeat (Program Orelse Auto)` (see section 4.11).

See also : chapter 12

4.11 Tacticals

We describe in this section how to combine the tactics provided by the system to write synthetic proof scripts called *tacticals*. The tacticals are built using tactic operators we present below.

4.11.1 `Idtac`

The constant `Idtac` is used as a *"pseudo tactic"* which leaves any goal unchanged.

4.11.2 `Do` *num tac*

This tactic operator repeats *num* times the tactic *tac*. It fails when it is not possible to repeat *num* times the tactic.

4.11.3 tac_1 `Orelse` tac_2

The tactical tac_1 `Orelse` tac_2 tries to apply tac_1 and, in case of a failure, applies tac_2. It associates to the left.

4.11.4 `Repeat` *tac*

This tactic operator repeats *tac* as long as it does not fail.

4.11.5 tac_1 `Then` tac_2

This tactic operator is a generalized composition for sequencing. The tactical tac_1 **`Then`** tac_2 applies tac_2 to all the subgoals generated by tac_1. **`Then`** associates to the left.

Variants :

1. tac_1 ; tac_2
   Is shorter syntax for tac_1 **`Then`** tac_2

4.11.6 tac; `[` tac_1 `|` ... `|` tac_n `]`

This tactic operator is a generalization of the precedent tactics operator. The tactical tac; `[` tac_1 `|` ... `|` tac_n `]` applies tac_i to the i-th subgoal generated by tac.

4.11.7 `Try` *tac*

This tactic operator applies tactic *tac*, and catches the possible failure of *tac*, it never fails.

Chapter 5

Other commands

5.1 Loadpath

5.1.1 `Pwd.`

This command calls the `pwd` UNIX command. It displays the current path.

5.1.2 `Cd` *string*.

This command calls the UNIX `cd` command. It changes the current directory according to *string* which can be any UNIX valid path.

Variants :

1. `Cd.`
   Is equivalent to `Cd "$COQTOP"`

5.1.3 `AddPath` *string*.

This command adds the path *string* to the current loadpath.

5.1.4 `DelPath` *string*.

This command removes the path *string* from the current loadpath.

5.1.5 `Print LoadPath.`

This command displays the current loadpath.

5.1.6 `Add ML Path` *string*.

This command adds the path *string* to the current Caml Light loadpath (see the command `Declare ML Module` in the section 5.3).

5.2 Loading files

When making a large development, one wants to divide it into several separate files. Then Coq offers the possibility of loading different parts of a whole development stored in separate files. Their contents will be loaded as if they were entered from the keyboard. This means that the loaded files are ASCII files containing sequences of commands for Coq's toplevel. This kind of file is called a *script* for Coq. The standard (and default) extension of Coq's script files is `.v`.

5.2.1 `Load` *ident*.

This command loads the file named *ident*`.v`, searching successively in each of the directories specified in the *loadpath*.

Variants :

1. `Load` *string*.
   Loads the file denoted by the string *string*, where *string* is any complete filename in the UNIX sense. Then the `~` and `..` abbreviations are allowed as well as shell variables. If no extension is specified, Coq will use the default extension `.v`

2. `Load Verbose` *ident*., `Load Verbose` *string*
   Display, while loading, the answers of Coq to each command (including tactics) contained in the loaded file
   See also : section 5.8.3

Error message :

1. `Can't find file` *ident* `on loadpath`

See also : section 5.1

5.3 Compiled files

This feature allows to build files for a quick loading. When loaded, the commands contained in a compiled file will not be *replayed*. In particular, proofs will not be replayed. This avoids a useless waste of time.

Remark : A module containing an open section cannot be compiled.

5.3.1 `Compile Module` *ident*.

This command is a new feature of coq V5.10. It loads the file *ident*`.v` and plays the script it contains. Declarations, definitions and proofs it contains are *"packaged"* in a compiled form : the *module* named *ident*. A file *ident*`.vo` is then created. The file *ident*`.v` is searched according to the current loadpath. The *ident*`.vo` is then written in the directory where *ident*`.v` was found.

Variants :

1. `Compile Module` *ident string*.

   Uses the file *string*`.v` or *string* if the previous one does not exist to build the module *ident*. In this case, *string* is any string giving a filename in the UNIX sense (see chapter 1).

2. `Compile Verbose Module` *ident*.
   Verbose version of Compile : shows the contents of the file being compiled

3. `Compile Verbose Module` *ident string*.
   Verbose version of Compile : shows the contents of the file being compiled.

Error message :

1. `You cannot open a module when there are things other than Modules and Imports in the context.`
   The only commands allowed before a Compile Module command are `Require`, `Read Module` and `Import`. The useful way to compile modules is in fact by the `coqc` command.

See also : sections 5.6.1, 5.1, chapter 13

5.3.2 `Read Module` *ident*.

Loads the module stored in the file *ident*, but does not open it : its contents is invisible to the user.

5.3.3 `Require` *ident*.

This command loads and opens (imports) the module stored in the file *ident*. If the module required has already been loaded, `Coq` displays the following warning : *ident* `already imported`.

If a module A contains a command `Require` B then the command `Require` A loads the module B but does not open it.

Variants :

1. `Require Export` *ident*.
   This command acts as `Require` *ident*. When it appears in another module $ident_0$, it specifies that the names defined by *ident* will be exported by $ident_0$ and consequentely visible after the command `Require` $ident_0$.

2. `Require Implementation` *ident*
   By now, is the same as `Require`.

Error message :

1. `Can't find file toto on loadpath`
   The command did not find the UNIX file `toto.vo`. Either `toto.v` exists but is not compiled or toto.vo is in a directory which is not in your `LoadPath`.

Remark : The `Require` of `Coq`V5.10 differs from the one of `Coq`V5.8. Indeed, in the previous version, `Require` acted as a clever `Load` for `.v` files ended with the `Provide`. In the new version, the command `Require` concerns only compiled modules.
See also : chapter 13

5.3.4 `Module` *ident*.

This command is used only by the `coqc` script, and is not intended to be used in another way. Therefore it is not documented here.

5.3.5 `Write Module` *ident*.

Similarly to the previous one, this command is used only by the `coqc` script, and is not intended to be used in another way. Therefore it is not documented here.
See also : chapter 13

5.3.6 `Print Modules.`

This command shows the currently loaded and currently opened (imported) modules.

5.3.7 `Declare ML Module` $string_1$ `..` $string_n$.

This commands loads the Caml Light compiled files $string_1 \dots string_n$ (dynamic link). It is mainly used to load tactics dynamically (see chapter 11). The files are searched into the current Caml Light loadpath (see the command `Add ML Path` in the section 5.1). Loading of Caml Light files is only possible under `coqtop` (not under `coq`).

5.4 States and Reset

5.4.1 `Reset` *ident*.

This command removes all the objects in the environment since *ident* was introduced, including *ident*. *ident* may be the name of a defined or declared object as well as the name of a section. One cannot reset over the name of a module or of an object inside a module.

Error message :

1. `cannot reset to a nonexistent object`

5.4.2 `Save State` *ident*.

Saves the current state of the development (mainly the defined objects) such that one can go back at this point if necessary.

Variants :

1. `Save State` *ident string*.
   Associates to the state of name `ident` the string *string* as a comment.

5.4.3 `Print States.`

Prints the names of the currently saved states with the associated comment. A state `Initial` is automatically built by the system.

5.4.4 `Restore State` *ident*.

Restores the set of known objects in the state *ident*.

Variants :

1. `Reset Initial.`
   Is equivalent to `Restore State Initial` and goes back to the initial state (like after the command `coqtop`).

5.4.5 `Remove State` *ident*.

Remove the state *ident* from the states list.

5.4.6 `Write States` *string*.

Writes the current list of states into a UNIX file *string*`.coq` for use in a further session. This file can be given as the `inputstate` argument of the commands `coqtop` and `coqc`. A command `Restore State` *ident* is necessary afterwards to choose explicitly which state to use (the default is to use `Initial`).

5.5 Displaying

5.5.1 `Print` *ident*.

This command displays on the screen informations about the declared or defined object *ident*.

Error message :

1. *ident* `not declared`

5.5.2 `Print All.`

This command displays informations about the current state of the environment, including sections and modules.

5.5.3 `Inspect` *num*.

This command displays the *num* last objects of the current environment, including sections and modules.

5.5.4 `Print Proof` *ident*.

Not yet documented.

5.5.5 `Print Section` *ident*.

Not yet documented

5.6 Requests to the environment

5.6.1 `Opaque` *ident*.

This command forbids the unfolding of the defined object *ident* by tactics using δ-conversion.
By default, `Theorem` and its alternatives are stamped as `Opaque`. This is to keep with the usual mathematical practice of *proof irrelevance*: what matters in a mathematical development is the sequence of lemma statements, not their actual proofs. This distinguishes lemmas from the usual defined constants, whose actual values are of course relevant in general.

See also : sections 4.4, 4.9, 3.1.3

5.6.2 `Transparent` *ident*.

This command is the converse of `Opaque`. By default, `Definition` and `Local` declare objects as `Transparent`.

Error message :

1. `Can not set transparent.`
   `It is a constant from a required module or a parameter.`

See also : sections 4.4, 4.9, 3.1.3

5.6.3 `Check` *ident*.

This command displays the type of *ident*.

Variants :

1. `Check` *term*.
   Displays the type of *term*.

5.6.4 `Eval` *term*.

This command gives the β-normal form of *term*.

5.6.5 `Compute` *term*.

This displays the $\beta\delta\iota$-normal form of *term*.

5.6.6 `Extraction` *ident*.

This command displays the $F\omega$-term extracted from *ident*. The name *ident* must refer to a defined constant or a theorem. The $F\omega$-term is extracted from the term defining *ident* when *ident* is a defined constant, or from the proof-term when *ident* is a theorem. The extraction is processed according to the distinguishing between `Set` and `Prop`; that is to say, between logical and computational content (see section 6.1.1).

Error message :

- `Non informative term`

See also : chapter 12

5.6.7 `Search` *ident*.

This command displays the name and type of all theorems of the current context whose statement's conclusion has the form (*ident* `t1 .. tn`). This command is very useful to remind the user of the name of library lemmas.

5.7 User's syntax facilities

We present in this section some syntaxic facilities which are new features of `Coq` V5.10. We will only sketch them here and refer the interested reader to chapter 9 for more details and examples.

5.7.1 `Syntactic Definition` *ident* `:=` *term*.

This command is a new feature of coq V5.10. It defines *ident* as an abbreviation with implicit arguments. Implicit arguments are denoted in *term* by `?` and they will have to be synthesized by the system.
Remark : Since it may contain dont't care variables `?`, the argument *term* of the `Syntactic Definition` cannot be typechecked at definition time. But each of its subsequent usages will be.
See also : chapter 9

5.7.2 `Syntax` $ident_1$ $ident_2$ `<<` *grammar-pattern* `>>`.

This command is a new feature of coq V5.10. It adresses the extensible grammar mechanism of `Coq`. It allows $ident_2$ to be parsed and pretty-printed as specified in *grammar-pattern*. Many examples of the `Syntax` command usage may be found in the `PreludeSyntax` file (see directory `$COQTOP/theories/INIT`).
See also : chapters 9, 10

5.7.3 `Grammar` $ident_1$ $ident_2$ `:=` *grammar-rule*.

This command is a new feature of `Coq`V5.10. It allows to give explicitly new grammar rules for parsing the user's own notation. It may be used instead of the two above syntaxic pragmas. But it can also be used by an advanced `Coq`'s user who programs his own tactics.
See also : chapters 9, 10, 4

5.7.4 `Token` *string*.

This command is a new feature of `Coq`V5.10. It allows the user to define a new token *string*, for instance to define new grammar rules through the commands `Grammar` or `Infix`. Lexical ambiguities are resolved according to the "longest match" rule. See the section 2.1 for more details.

5.7.5 `Infix` *num string ident*.

This command is a new feature of CoqV5.10. It declares a prefix operator *ident* as infix, with the syntax *term string term*. *num* is the precedence associated to the operator; it must lie between 6 and 9. The infix operator *string* associates to the right. *string* must be a legal token. Both grammar and pretty-print rules are automatically generated for *string*.

5.8 Miscellaneous

5.8.1 `Quit.`

This command permits to quit Coq.

5.8.2 `Drop.`

This command permits to leave Coq temporarily and enter the caml-light toplevel. The caml command `go();;` will allow subsequently to return to Coq's toplevel in the same state. This is used mostly as a debug facility by Coq'implementors and does not concern the casual user.

5.8.3 `Begin Silent.`

This command turns off the normal displaying.

5.8.4 `End Silent.`

This command turns the normal display on.

Chapter 6

The Calculus of Inductive Constructions

The underlying formal language of Coq is the *Calculus of Inductive Constructions* (CIC in short). It is a formulation of type theory including the possibility of inductive constructions.
One important feature of type theories is that they manipulate two sorts of objects, namely terms and types. Types describe classes to which terms can belong. Any object handled in the formalism must *explicitly* belong to a type. For instance, the statement *"for all x, P"* is not allowed in type theory; you must say instead : *"for all x belonging to T, P"*. The expression *"x belonging to T"* is written *"x:T"*. One also says : *"x is of type T"*.

The purpose of this part is to precisely present the typing rules of the system and introduce various theoretical notions that must be understood in order to use the Coq commands.

An introduction to various related typed lambda-calculi can be found in [3]. A formal study of the Calculus of Inductive Constructions can be found in [74].

6.1 The terms

In most type theories, one usually makes a syntactic distinction between types and terms. This is not the case for CIC which defines both types and terms in the same syntactical structure. This is because the type-theory itself forces terms and types to be defined in a mutual recursive way and also because similar constructions can be applied to both terms and types and consequently can share the same syntactic structure.

For instance the type of functions will have several meanings. Assume nat is the type of natural numbers then $\mathsf{nat} \rightarrow \mathsf{nat}$ is the type of functions from nat to nat, $\mathsf{nat} \rightarrow \mathsf{Prop}$ is the type of unary predicates over the natural numbers. For instance $[x : \mathsf{nat}](x = x)$ will represent a predicate P, informally written in mathematics $P(x) \equiv x = x$. If P has type $\mathsf{nat} \rightarrow \mathsf{Prop}$, $(P\ x)$ is a proposition, furthermore $(x : \mathsf{nat})(P\ x)$ will represent the type of functions which associate to each natural number n an object of type $(P\ n)$ and consequently represent proofs of the formula "$\forall x.P(x)$".

6.1.1 Sorts

Types are seen as terms of the language and then should belong to another type. The type of a type is always a constant of the language called a sort.

The two basic sorts in the language of CIC are Set and Prop.

The sort Prop intends to be the type of logical propositions. If M is a logical proposition then it denotes a class, namely the class of terms representing proofs of M. An object m belonging to M witnesses the fact that M is true. An object of type Prop is often called a *predicate*.

The sort Set intends to be the type of usual sets such as booleans, naturals, lists etc. Objects of type Set are said to be *constructive families*.

These sorts themselves can be manipulated as ordinary terms. Consequently sorts also should be given a type. Because assuming simply that Set has type Set leads to an inconsistent theory, we have infinitely many sorts in the language of CIC . These are, in addition to Set and Prop two hierarchies of universes Type(i), Typeset(i) for any integer i. We call $\mathcal{S}$ the set of sorts which is defined by :

$$\mathcal{S} \equiv \{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}(i), \mathsf{Typeset}(i) | i \in \mathbb{N}\}$$

The sorts enjoy the following properties : Prop:Type(0), Set:Typeset(0), Type(i):Type($i+1$) and Typeset(i):Typeset($i+1$).

The user will never mention explicitly the index i when referring to the universe Type(i). One only writes Type or Typeset. The system itself generates for each instance of Type or Typeset a new index for the universe and checks that the constraints between these indexes can be solved. From the user point of view we consequently have Type :Type and Typeset:Typeset.

We shall precise in the typing rules the constraints between the indexes.

Remark. The existence of two distinct hierarchies of sorts Prop and Type versus Set and Typeset is related to the mechanism of extracting programs from proofs. The system will never extract any information from objects (called logical or non-computational) whose type belongs to Prop or Type. At the opposite any object A belonging to Set is interpreted in a constructive way as a specification and any object a belonging to A can be interpreted as a program which is correct with respect to the specification A. From the typing point of view, the system prevents the use of a non-computational object in the construction of a computational object. This insures that it is always possible to erase the non-computational part in a consistent way.

6.1.2 Constants

Besides the sorts, the language also contains constants denoting objects in the environment. These constants may denote previously defined objects but also objects related to inductive definitions (either the type itself or one of its constructors or destructors).

Remark. In other presentations of CIC, the inductive objects are not seen as external declarations but as first-class terms. Usually the definitions are also completely ignored. This is a nice theoretical point of view but not so practical. An inductive definition is specified by a possibly huge set of declarations, clearly we want to share this specification among the various inductive objects and not to duplicate it. So the specification should exist somewhere and the various objects should refer to it. We choose one more level of indirection where the objects are just represented as constants and the environment gives the information on the kind of object the constant refers to.

Our inductive objects will be manipulated as constants declared in the environment. This roughly corresponds to the way they are actually implemented in the Coq system. It is simple to map this presentation in a theory where inductive objects are represented by terms.

6.1.3 Language

Types. Roughly speaking types can be separated into atomic and composed types.

An atomic type of the *Calculus of Inductive Constructions* is either a sort or is built from a type variable or an inductive definition applied to some terms.

A composed type will be a product $(x : T)U$ with T and U two types.

Terms. A term is either a type or a term variable or a term constant of the environment.

As usual in λ-calculus, we combine objects using abstraction and application.

More precisely the language of the *Calculus of Inductive Constructions* is built with the following rules :

1. the sorts Set, Prop, Type, Typeset are terms.
2. constants of the environment are terms.
3. variables are terms.
4. if x is a variable and T, U are terms then $(x : T)U$ is a term. If x occurs in U, $(x : T)U$ reads as *"for all x of type T, U"*. As U depends on x, one says that $(x : T)U$ is a *dependent product*. If x doesn't occurs in U then $(x : T)U$ reads as *"if T then U"*. A non dependent product can be written : $T \rightarrow U$.
5. if x is a variable and T, U are terms then $[x : T]U$ is a term. This is a notation for the λ-abstraction of λ-calculus [5]. The term $[x : T]U$ is a function which maps elements of T to U.
6. if T and U are terms then $(T\ U)$ is a term. The term $(T\ U)$ reads as *"T applied to U"*.

Notations. Application associates to the left such that $(t\ t_1 \dots t_n)$ represents $(\dots(t\ t_1)\dots t_n)$. The products and arrows associates to the right such that $(x : A)B \rightarrow C \rightarrow D$ represents $(x : A)(B \rightarrow (C \rightarrow D))$. One uses sometimes $(x, y : A)B$ or $[x, y : A]B$ to denote the abstraction or product of several variables of the same type. The equivalent formulation is $(x : A)(y : A)B$ or $[x : A][y : A]B$

Free variables. The notion of free variables is defined as usual. In the expressions $[x : T]U$ and $(x : T)U$ the occurrences of x in U are bound. They are represented by de Bruijn indexes in the internal structure of terms.

Substitution. The notion of substituting a term T to free occurrences of a variable x in a term U is defined as usual. The resulting term will be written $U\{x/T\}$.

6.2 Typed terms

As objects of type theory, terms are subjected to *type discipline*. The well typing of a term depends on a set of declarations of variables we call a *context*. A context Γ is written $[x_1 : T_1; ..; x_n : T_n]$ where the x_i's are distinct variables and the T_i's are terms. If Γ contains some $x : T$, we write $(x : T) \in \Gamma$ and also $x \in \Gamma$. Contexts must be themselves *well formed*. The notation $\Gamma :: (y : T)$ denotes the context $[x_1 : T_1; ..; x_n : T_n; y : T]$. The notation $[]$ denotes the empty context.

We define the inclusion of two contexts Γ and Δ (written as $\Gamma \subset \Delta$) as the property, for all variable x and type T, if $(x : T) \in \Gamma$ then $(x : T) \in \Delta$. We write $|\Delta|$ for the length of the context Δ which is n if Δ is $[x_1 : T_1; ..; x_n : T_n]$.

A variable x is said to be free in Γ if Γ contains a declaration $y : T$ such that x is free in T.

Environment. Because we are manipulating constants, we also need to consider an environment E. We shall give afterwards the rules for introducing new objects in the environment. For the typing relation of terms, it is enough to introduce two notions. One which says if a name is defined in the environment we shall write $c \in E$ and the other one which gives the type of this constant in E. We shall write $(c : T) \in E$.

In the following, we assume E is a valid environment. We define simultaneously two judgments. The first one $E[\Gamma] \vdash t : T$ means the term t is well-typed and has type T in the environment E and context Γ. The second judgment $\mathcal{WF}(E)[\Gamma]$ means that the environment E is well-formed and the context Γ is a valid context in this environment. It also means a third property which makes sure that any constant in E was defined in an environment which is included in Γ *.

A term t is well typed in an environment E iff there exists a context Γ and a term T such that the judgment $E[\Gamma] \vdash t : T$ can be derived from the following rules.

W-E
$$\mathcal{WF}([])[[]]$$

W-s
$$\frac{E[\Gamma] \vdash T : s \quad s \in \mathcal{S} \quad x \notin \Gamma \cup E}{\mathcal{WF}(E)[\Gamma :: (x : T)]}$$

Ax
$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{Prop} : \mathsf{Type}(p)} \qquad \frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{Set} : \mathsf{Typeset}(q)}$$

$$\frac{\mathcal{WF}(E)[\Gamma] \quad i < j}{E[\Gamma] \vdash \mathsf{Type}(i) : \mathsf{Type}(j)} \qquad \frac{\mathcal{WF}(E)[\Gamma] \quad i < j}{E[\Gamma] \vdash \mathsf{Typeset}(i) : \mathsf{Typeset}(j)}$$

Var
$$\frac{\mathcal{WF}(E)[\Gamma] \quad (x : T) \in \Gamma}{E[\Gamma] \vdash x : T}$$

Const
$$\frac{\mathcal{WF}(E)[\Gamma] \quad (c : T) \in E}{E[\Gamma] \vdash c : T}$$

Prod
$$\frac{E[\Gamma] \vdash T : s_1 \quad E[\Gamma :: (x : T)] \vdash U : s_2 \quad s_1 \in \{\mathsf{Prop}, \mathsf{Set}\} \text{ or } s_2 \in \{\mathsf{Prop}, \mathsf{Set}\}}{E[\Gamma] \vdash (x : T)U : s_2}$$

*This requirement could be relaxed if we instead introduced an explicit mechanism for instantiating constants. At the external level, the Coq engine works accordingly to this view that all the definitions in the environment were built in a sub-context of the current context.

$$\frac{E[\Gamma] \vdash T : t'(i) \quad E[\Gamma :: (x:T)] \vdash U : t(j) \quad t, t' \in \{\mathsf{Type}, \mathsf{Typeset}\} \quad i \leq k \quad j \leq k}{E[\Gamma] \vdash (x:T)U : t(k)}$$

Lam $$\frac{E[\Gamma] \vdash (x:T)U : s \quad E[\Gamma :: (x:T)] \vdash t : U}{E[\Gamma] \vdash [x:T]t : (x:T)U}$$

App $$\frac{E[\Gamma] \vdash t : (x:U)T \quad E[\Gamma] \vdash u : U}{E[\Gamma] \vdash (t\ u) : T\{x/u\}}$$

6.3 Conversion rules

β-reduction. We want to be able to identify some terms as we can identify the application of a function to a given argument with its result. For instance the identity function over a given type T can be written $[x:T]x$. We want to identify any object a (of type T) with the application $([x:T]x\ a)$. We define for this a *reduction* (or a *conversion*) rule we call β :

$$([x:T]t\ u) \triangleright_\beta t\{x/u\}$$

We say that $t\{x/u\}$ is the *β-contraction* of $([x:T]t\ u)$ and, conversely, that $([x:T]t\ u)$ is the *β-expansion* of $t\{x/u\}$.

According to β-reduction, terms of the *Calculus of Inductive Constructions* enjoy some fundamental properties such as confluence, strong normalization, subject reduction. These results are theoretically of great importance but we will not detail them here and refer the interested reader to [13].

ι-reduction. A specific conversion rule is associated to the inductive objects in the environment. We shall give later on (section 6.5.4) the precise rules but it just says that a destructor applied to an object built from a constructor behaves as expected. This reduction is called ι-reduction and is more precisely studied in [66, 74].

δ-reduction. In the environment we also have constants representing abbreviations for terms. It is legal to identify a constant with its value. This reduction will be precised in section 6.4.1 where we define well-formed environments. This reduction will be called δ-reduction.

Convertibility. Let us write $t \triangleright u$ for the relation t reduces to u with one of the previous reduction β, ι or δ.

We say that two terms t_1 and t_2 are *convertible* (or *equivalent)* iff there exists a term u such that $t_1 \triangleright \ldots \triangleright u$ and $t_2 \triangleright \ldots \triangleright u$. We note $t_1 =_{\beta\delta\iota} t_2$.

The convertibility relation allows to introduce a new typing rule which says that two convertible well-formed types have the same inhabitants.

At the moment, we did not take into account one rule between universes which says that any term in a universe of index i is also a term in the universe of index $i+1$. This property is included into the conversion rule by extending the equivalence relation of convertibility into an order inductively defined by :

1. if $M =_{\beta\delta\iota} N$ then $M \leq_{\beta\delta\iota} N$,

2. if $i \leq j$ then $\mathsf{Type}(i) \leq_{\beta\delta\iota} \mathsf{Type}(j)$ and $\mathsf{Typeset}(i) \leq_{\beta\delta\iota} \mathsf{Typeset}(j)$,

3. if $T =_{\beta\delta\iota} U$ and $M \leq_{\beta\delta\iota} N$ then $(x:T)M \leq_{\beta\delta\iota} (x:U)N$.

The conversion rule is now exactly :

Conv $$\frac{E[\Gamma] \vdash U : S \quad E[\Gamma] \vdash t : T \quad T \leq_{\beta\delta\iota} U}{E[\Gamma] \vdash t : U}$$

η-conversion. An other important rule is the η-conversion. It is to identify terms over a dummy abstraction of a variable followed by an application of this variable. Let T be a type, t be a term in which the variable x doesn't occurs free. We have

$$[x:T](t\ x) \triangleright t$$

Indeed, as x doesn't occurs free in t, for any u one applies to $[x:T](t\ x)$, it β-reduces to $(t\ u)$. So $[x:T](t\ x)$ and t can be identified.
Remark : The η-reduction is not taken into account in the convertibility rule of Coq.

Normal form. A term which cannot be any more reduced is said to be in *normal form*. There are several ways (or strategies) to apply the reduction rule. Among them, we have to mention the *head reduction* which will play an important role (see chapter 4). Any term can be written as $[x_1:T_1]\ldots[x_k:T_k](t_0\ t_1\ldots t_n)$ where t_0 is not an application. We say then that t_0 is the *head of* t. If we assume that t_0 is $[x:T]u_0$ then one step of β-head reduction of t is :

$$[x_1:T_1]\ldots[x_k:T_k]([x:T]u_0\ t_1\ldots t_n) \ \triangleright\ [x_1:T_1]\ldots[x_k:T_k](u_0\{x/t_1\}\ t_2\ldots t_n)$$

Iterating the process of head reduction until the head of the reduced term is no more an abstraction leads to the *β-head normal form* of t :

$$t \triangleright \ldots \triangleright [x_1:T_1]\ldots[x_k:T_k](v\ u_1\ldots u_m)$$

where v is not an abstraction (nor an application). Note that the head normal form must not be confused with the normal form since some u_i can be reducible.

Similar notions of head-normal forms involving δ and ι reductions or any combination of those can also be defined.

6.4 Definitions in environments

We now give the rules for manipulating objects in the environment. Because a constant can depend on previously introduced constants, the environment will be an ordered list of declarations. When specifying an inductive definition, several objects will be introduced at the same time. So any object in the environment will define one or more constants.

In this presentation we introduce two different sorts of objects in the environment. The first one is ordinary definitions which give a name to a particular well-formed term, the second one is inductive definitions which introduce new inductive objects.

6.4.1 Rules for definitions

Adding a new definition. The simplest objects in the environment are definitions which can be seen as one possible mechanism for abbreviation.

A definition will be represented in the environment as $\mathsf{Def}(\Gamma)(c := t : T)$ which means that c is a constant which is valid in the context Γ whose value is t and type is T.

δ-reduction. If $\mathsf{Def}(\Gamma)(c := t : T)$ is in the environment E then in this environment the δ-reduction $c \triangleright_\delta t$ is introduced.

The rule for adding a new definition is simple :

Def
$$\frac{E[\Gamma] \vdash t : T \quad c \notin E \cup \Gamma}{\mathcal{WF}(E; \mathsf{Def}(\Gamma)(c := t : T))[\Gamma]}$$

6.4.2 Derived rules

From the original rules of the type system, one can derive new rules which change the context of definition of objects in the environment. Because these rules correspond to elementary operations in the Coq engine used in the discharge mechanism at the end of a section, we state them explicitly.

Mechanism of substitution. One rule which can be proved valid, is to replace a term c by its value in the environment. As we defined the substitution of a term for a variable in a term, one can define the substitution of a term for a constant. One easily extends this substitution to contexts and environments.

Substitution Property :
$$\frac{\mathcal{WF}(E; \mathsf{Def}(\Gamma)(c := t : T); F)[\Delta]}{\mathcal{WF}(E; F\{c/t\})[\Delta\{c/t\}]} \quad .$$

Abstraction. One can modify the context of definition of a constant c by abstracting a constant with respect to the last variable x of its defining context. For doing that, we need to check that the constants appearing in the body of the declaration do not depend on x, we need also to modify the reference to the constant c in the environment and context by explicitly applying this constant to the variable x. Because of the rules for building environments and terms we know the variable x is available at each stage where c is mentioned.

Abstracting property :
$$\frac{\mathcal{WF}(E; \mathsf{Def}(\Gamma :: (x : U))(c := t : T); F)[\Delta] \quad \mathcal{WF}(E)[\Gamma]}{\mathcal{WF}(E; \mathsf{Def}(\Gamma)(c := [x : U]t : (x : U)T); F\{c/(c\ x)\})[\Delta\{c/(c\ x)\}]}$$

Pruning the context. We said the judgment $\mathcal{WF}(E)[\Gamma]$ means that the defining contexts of constants in E are included in Γ. If one abstracts or substitutes the constants with the above rules then it may happen that the context Γ is now bigger than the one needed for defining the constants in E. Because defining contexts are growing in E, the minimum context needed for defining the constants in E is the same as the one for the last constant. One can consequently derive the following property.

Pruning property :
$$\frac{\mathcal{WF}(E; \mathsf{Def}(\Delta)(c := t : T))[\Gamma]}{\mathcal{WF}(E; \mathsf{Def}(\Delta)(c := t : T))[\Delta]} \quad .$$

6.5 Inductive Definitions

A (possibly mutual) inductive definition is specified by giving the names and the type of the inductive sets or families to be defined and the names and types of the constructors of the inductive predicates. An inductive declaration in the environment can consequently be represented with two contexts (one for inductive definitions, one for constructors).

Stating the rules for inductive definitions in their general form needs quite tedious definitions. We shall try to give a concrete understanding of the rules by precising them on running examples. We take as examples the type of natural numbers, the type of parameterized lists over a type A, the relation which state that a list has some given length and the mutual inductive definition of trees and forests.

6.5.1 Representing an inductive definition

We have to slightly complicate this representation with the two contexts for definitions and constructors. An inductive declaration can be done in whatever context we want, consequently we have, as for definitions, to keep track of the defining context of variables. But of course we may also want to use a definition in an alternative context. For definition this is done using the abstraction/application mechanism. Just doing a generalization on the types in the inductive declaration is not satisfactory because it loses some information that is useful when deriving the destructor operator[†]. Consequently we keep track of two contexts, one is the defining context and the other one is the abstracted context also called the context of *parameters* of the definition. An inductive definition can be used in whatever context which extends the defining contexts, and will be *explicitly* applied to terms which instantiate the parameters.

We write $\mathsf{Ind}(\Gamma)[\Gamma_P](\ \Gamma_I := \Gamma_C\ )$ an inductive definition valid in a context Γ with parameters Γ_P, a context of definitions Γ_I and a context of constructors Γ_C.

The occurrences of the variables of Γ_P in the contexts Γ_I and Γ_C are bound.

Examples. The inductive declaration for the type of natural numbers will be :

$$\mathsf{Ind}([])[[]](\ \mathsf{nat} : \mathsf{Set} := \mathsf{O} : \mathsf{nat}, \mathsf{S} : \mathsf{nat} \to \mathsf{nat}\ )$$

The declaration for parameterized lists is :

$$\mathsf{Ind}([])[A : \mathsf{Set}](\ \mathsf{list} : \mathsf{Set} := \mathsf{nil} : \mathsf{list}, \mathsf{cons} : A \to \mathsf{list} \to \mathsf{list}\ )$$

The declaration for lists of length n is :

$$\begin{array}{ll}\mathsf{Ind}([])[A : \mathsf{Set}] & (\mathsf{Length} : (\mathsf{list}\ A) \to \mathsf{nat} \to \mathsf{Prop} := \\ & \quad \mathsf{Lnil} : (\mathsf{Length}\ (\mathsf{nil}\ A)\ \mathsf{O}) \\ & \quad \mid \mathsf{Lcons} : (a : A)(l : (\mathsf{list}\ A))(n : \mathsf{nat})(\mathsf{Length}\ l\ n) \to \\ & \qquad\qquad (\mathsf{Length}\ A\ (\mathsf{cons}\ A\ a\ l)\ (\mathsf{S}\ n)))\end{array}$$

[†]This problem is a bit technical and probably not yet well-understood at the theoretical level. The intuition is that an instance of an inductive definition does not necessarily have an inductive structure, but if we first abstract the inductive declaration and then reinstantiate it then we want to recover the initial structure. That is why we keep this context of parameters as large as possible.

The declaration for a mutual inductive definition of forests and trees is :

$$\mathsf{Ind}([])[[]](\ \mathsf{tree} : \mathsf{Set}, \mathsf{forest} : \mathsf{Set} := \mathsf{node} : \mathsf{forest} \to \mathsf{tree}, \mathsf{emptyf} : \mathsf{forest}, \mathsf{consf} : \mathsf{tree} \to \mathsf{forest} \to \mathsf{forest}\ )$$

These representations are the ones obtained as the result of the Coq declaration :

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

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

Coq < Inductive Length [A:Set] : (list A) -> nat -> Prop :=
Coq <     Lnil : (Length A (nil A) O)
Coq <  | Lcons : (a:A)(l:(list A))(n:nat)
Coq <            (Length A l n)->(Length A (cons A a  l) (S n)).

Coq < Mutual Inductive tree : Set := node : forest -> tree
Coq < with forest : Set := emptyf : forest | consf : tree -> forest -> forest.
```

6.5.2 Types of inductive objects

We have to give the type of constants in an environment E which contains an inductive declaration.

Ind-Const Assuming Γ_P is $[p_1 : P_1; \ldots; p_r : P_r]$, Γ_I is $[I_1 : A_1; \ldots; I_k : A_k]$, and Γ_C is $[c_1 : C_1; \ldots; c_n : C_n]$,

$$\frac{\mathsf{Ind}(\Gamma)[\Gamma_P](\ \Gamma_I := \Gamma_C\ ) \in E \quad j = 1 \ldots k}{(I_j : (p_1 : P_1) \ldots (p_r : P_r) A_j) \in E}$$

$$\frac{\mathsf{Ind}(\Gamma)[\Gamma_P](\ \Gamma_I := \Gamma_C\ ) \in E \quad i = 1..n}{(c_i : (p_1 : P_1) \ldots (p_r : P_r) C_i\{I_j/(I_j\ p_1 \ldots p_r)\}_{j=1\ldots k}) \in E}$$

Example. We have $(\mathsf{list} : \mathsf{Set} \to \mathsf{Set}), (\mathsf{cons} : (A : \mathsf{Set}) A \to (\mathsf{list}\ A) \to (\mathsf{list}\ A))$, $(\mathsf{Length} : (A : \mathsf{Set})(\mathsf{list}\ A) \to \mathsf{nat} \to \mathsf{Prop})$, $\mathsf{tree} : \mathsf{Set}$ and $\mathsf{forest} : \mathsf{Set}$.

From now on, we write list_A instead of $(\mathsf{list}\ A)$ and Length_A for $(\mathsf{Length}\ A)$.

Parameters. The parameters introduce a distortion between the inside specification of the inductive declaration where parameters are supposed to be instantiated (this representation is appropriate for checking the correctness or deriving the destructor principle) and the outside typing rules where the inductive objects are seen as objects abstracted with respect to the parameters.

In the definition of list or Length A is a parameter because what is effectively inductively defined is list_A or Length_A for a given A which is constant in the type of constructors. But when we define $(\mathsf{Length}_A\ l\ n)$, l and n are not parameters because the constructors manipulate different instances of this family.

6.5.3 Well-formed inductive definitions

We cannot accept any inductive declaration because some of them lead to inconsistent systems. We restrict ourselves to definitions which satisfy a syntactic criterion of positivity. Before giving the formal rules, we need a few definitions :

Definitions A type T is an *arity of sort* s if it is the sort s or a product $(x:T)U$ with U an arity of sort s. (For instance $A \rightarrow \mathsf{Set}$ or $(A:\mathsf{Prop})A \rightarrow \mathsf{Prop}$ are arities of sort respectively Set and Prop).

A *type of constructor of* I is either a term $(I\ t_1 \dots\ t_n)$ or $(x:T)C$ with C a *type of constructor of* I. It will be said to *satisfy the positivity condition* with respect to a constant X if X does not occur in t_i and occurs only strictly positively in each domain of product T.

The constant X *occurs strictly positively* in $(X\ t_1 \dots\ t_n)$ if it does not occur in t_i and occurs strictly positively in $(x:T)U$ if it does not occur in T and occurs strictly positively in U.

Example For instance X occurs strictly positively in $A \rightarrow X$ but not in $X \rightarrow A$ or $(X \rightarrow A) \rightarrow A$ or $X * A$ or $(\texttt{list}\ X)$ assuming the notion of product and lists were already defined. In the last two cases it is easy to define an equivalent (possibly mutual inductive) definition which enjoys the positivity condition.

Correctness rules. We shall now describe the rules allowing the introduction of a new inductive definition.

W-Ind Let E be an environment and $\Gamma, \Gamma_P, \Gamma_I, \Gamma_C$ are contexts such that Γ_I is $[I_1 : A_1; \dots; I_k : A_k]$ and Γ_C is $[c_1 : C_1; \dots; c_n : C_n]$.

$$\frac{(E[\Gamma;\Gamma_P] \vdash A_j : s'_j)_{j=1\dots k} \quad (E[\Gamma;\Gamma_P;\Gamma_I] \vdash C_i : s_{p_i})_{i=1\dots n}}{\mathcal{WF}(E;\mathsf{Ind}(\Gamma)[\Gamma_P](\ \Gamma_I := \Gamma_C\ ))[\Gamma]}$$

providing the following side conditions hold :

- $k > 0$, I_j, c_i are different names for $j = 1 \dots k$ and $i = 1 \dots n$,
- for $j = 1 \dots k$ we have A_j is an arity of sort s_j and $I_j \notin \Gamma \cup E$,
- for $i = 1 \dots n$ we have C_i is a type of constructor of I_{p_i} which satisfies the positivity condition for $I_1 \dots I_k$ and $c_i \notin \Gamma \cup E$.

One can remark that there is a constraint between the sort of the arity of the inductive type and the sort of the type of its constructors which will always be satisfied for impredicative sorts (Prop or Set) but may generate constraints between universes.

Recursive arguments of constructors. From the specification of the inductive definition, one can easily define a notion of *recursive arguments* for a constructor. Namely when looking at the type of c which has the shape $(p_1 : P_1) \dots (p_r : P_r)(x_1 : T_1) \dots (x_r : T_r)(I_j\ p_1 \dots p_r\ t_1 \dots t_s)$ the recursive arguments will correspond to T_i in which one of the I_l occurs.

One needs to define carefully this notion over the abstracted type, because as soon as we instantiate a constructor with paramaters one could by just inspecting the instantiated type find more recursive arguments than the real ones.

For instance we can perfectly define $(\mathsf{list}\ (\mathsf{list}\ \mathsf{nat}))$, then $(\mathsf{cons}\ (\mathsf{list}\ \mathsf{nat}))$ has type $(\mathsf{list}\ \mathsf{nat}) \rightarrow (\mathsf{list}\ (\mathsf{list}\ \mathsf{nat})) \rightarrow (\mathsf{list}\ (\mathsf{list}\ \mathsf{nat}))$ even if list occurs in the type of the first argument we do not want to consider it as part of the structural induction on lists.

6.5.4 Destructors

The specification of inductive definitions with arities and constructors is quite natural. But we still have to say how to use an object in an inductive type.

This problem is rather delicate. There are actually several different ways to do that. Some of them are logically equivalent but not always equivalent from the computational point of view or from the user point of view.

From the computational point of view, we want to be able to define a function whose domain is an inductively defined type by using a combination of case analysis over the possible constructors of the object and recursion.

Because we need to keep a consistent theory and also we prefer to keep a strongly normalising reduction, we cannot accept any sort of recursion (even terminating). So the basic idea is to restrict ourselves to primitive recursive functions and functionals.

For instance, assuming a parameter $A : \mathsf{Set}$ exists in the context, we want to build a function lgth of type $\mathsf{list}_A \rightarrow \mathsf{nat}$ which computes the length of the list, so such that $(\mathsf{lgth}\ \mathsf{nil}) = \mathsf{O}$ and $(\mathsf{lgth}\ (\mathsf{cons}\ A\ a\ l)) = (\mathsf{S}\ (\mathsf{lgth}\ l))$. We want these equalities to be recognized implicitely and taken into account in the conversion rule.

From the logical point of view, we have built a type family by giving a set of constructors. We want to capture the fact that we do not have any other way to build an object in this type. So when trying to prove a property $(P\ m)$ for m in an inductive definition it is enough to enumerate all the cases where m starts with a different constructor.

In case the inductive definition is effectively a recursive one, we want to capture the extra property that we have built the smallest fixed point of this recursive equation. This says that we are only manipulating finite objects. This analysis provides induction principles.

For instance, in order to prove $(l : \mathsf{list}_A)(\mathsf{Length}_A\ l\ (\mathsf{lgth}\ l))$ it is enough to prove :
$(\mathsf{Length}_A\ \mathsf{nil}\ (\mathsf{lgth}\ \mathsf{nil}))$ and
$(a : A)(l : \mathsf{list}_A)(\mathsf{Length}_A\ l\ (\mathsf{lgth}\ l)) \rightarrow (\mathsf{Length}_A\ (\mathsf{cons}\ A\ a\ l)\ (\mathsf{lgth}\ (\mathsf{cons}\ A\ a\ l)))$.

which given the conversion equalities satisfied by lgth is the same as proving : $(\mathsf{Length}_A\ \mathsf{nil}\ \mathsf{O})$ and
$(a : A)(l : \mathsf{list}_A)(\mathsf{Length}_A\ l\ (\mathsf{lgth}\ l)) \rightarrow (\mathsf{Length}_A\ (\mathsf{cons}\ A\ a\ l)\ (\mathsf{S}\ \mathsf{lgth}\ l))$.

One conceptually simple way to do that, following the basic scheme proposed by Martin-Löf in his Intuitionistic Type Theory, is to introduce for each inductive definition an elimination operator. At the logical level it is a proof of the usual induction principle and at the computational level it implements a generic operator for doing primitive recursion over the structure.

But this operator is rather tedious to implement and use. We choose in this version of Coq to factorize the operator for primitive recursion into two more primitive operations as was first suggested by Th. Coquand in [16]. One is the definition by case analysis. The second one is a definition by guarded fixpoints.

The `Case`... `of` ... `end` construction.

The basic idea of this destructor operation is that we have an object m in an inductive type I and we want to prove a property $(P\ m)$ which in general depends on m. For this, it is enough to prove the property for $m = (c_i\ u_1 \dots u_p)$ for each constructor of I.

This proof will be denoted by a generic term :

$$<\!P\!>\mathsf{Case}\ m\ \mathsf{of}\ f_1 \dots f_n\ \mathsf{end}$$

If in this expression m is a term built from a constructor $(c_i\ u_1 \dots u_p)$ then the expression will behave as it is specified with i-th branch and will reduce to $(f_i\ u_1 \dots u_p)$ according to the ι-reduction.

This is the basic idea which is generalized to the case where I is an inductively defined n-ary relation (in which case the property P to be proved will be a $n+1$-ary relation).

Non-dependent elimination. When defining a function by case analysis, we build an object of type $I \to C$ and the minimality principle on an inductively defined logical predicate of type $A \to \mathsf{Prop}$ is often used to prove a property $(x : A)(I\ x) \to (C\ x)$. This is a particular case of the dependent principle that we stated before with a predicate which does not depend explicitly on the object in the inductive definition.

For instance, a function testing whether a list is empty can be defined as :

$$[l : \mathsf{list}_A]<\![H : \mathsf{list}_A]\mathsf{bool}\!>\mathsf{Case}\ l\ \mathsf{of}\ \mathsf{true}\ [a : A][m : \mathsf{list}_A]\mathsf{false}\ \mathsf{end}$$

Remark. In the system the expression, without mentioning the dummy abstraction can also be directly interpreted. $<\!\mathsf{bool}\!>\mathsf{Case}\ l\ \mathsf{of}\ \mathsf{true}\ [a : A][m : \mathsf{list}_A]\mathsf{false}\ \mathsf{end}$

Allowed elimination sorts. An important question for building the typing rule for `Case` is what can be the type of P with respect to the type of the inductive definitions.

Remembering that the elimination builds an object in $(P\ m)$ from an object in I it is clear that we cannot allow any combination.

For instance we cannot in general have I is a non-computational object and P is a computational family. But the other way is safe with respect to our interpretation we can have I a computational object and P a non-computational one, it just corresponds to proving a logical property of a computational object.

Also if I is in one of the sorts $\{\mathsf{Prop}, \mathsf{Set}\}$, one cannot in general allow an elimination over a bigger sort such as Type or $\mathsf{Typeset}$. But this operation is safe whenever I is a *small inductive* type, which means that all the types of constructors of I are small with the following definition :
$(I\ t_1 \dots t_s)$ is a *small type of constructor* and $(x : T)C$ is a small type of constructor if C is and if T has type Prop or Set.

We call this particular elimination which gives the possibility to compute a type by induction on the structure of a term, a *strong elimination*.

We define now a relation $[I : A|B]$ between an inductive definition I of type A, an arity B which says that an object in the inductive definition I can be eliminated for proving a property P of type B.

The $[I : A|B]$ is defined as the smallest relation satisfying the following rules :

Prod $$\frac{[(I\ x) : A'|B']}{[I : (x : A)A'|(x : A)B']}$$

Prop $$[I : \mathsf{Prop}|I \to \mathsf{Prop}] \qquad \frac{I \text{ is a singleton definition}}{[I : \mathsf{Set}|I \to \mathsf{Set}]}$$

Set	$\dfrac{s \in \{\mathsf{Prop},\mathsf{Set}\}}{[I:\mathsf{Set}\vert I \rightarrow s]}$	$\dfrac{I \text{ is a small inductive definition} \quad t \in \{\mathsf{Type},\mathsf{Typeset}\}}{[I:\mathsf{Set}\vert I \rightarrow t(i)]}$
Type		$\dfrac{s \in \{\mathsf{Prop},\mathsf{Set},\mathsf{Type}(j)\vert j \leq i\}}{[I:\mathsf{Type}(i)\vert I \rightarrow s]}$
Typeset		$\dfrac{s \in \{\mathsf{Prop},\mathsf{Set},\mathsf{Type}(j),\mathsf{Typeset}(j)\vert j \leq i\}}{[I:\mathsf{Typeset}(i)\vert I \rightarrow s]}$

Notations. We write $[I|B]$ for $[I:A|B]$ where A is the type of I.

Warning : strong elimination In previous versions of Coq, for a small inductive definition, only the non-informative strong elimination on Type was allowed, because strong elimination on Typeset was not compatible with the current extraction procedure. In this version, strong elimination on Typeset is accepted but a dummy element is extracted from it and may generate problems if extracted terms are explicitly used such as in the `Program` tactic or when extracting ML programs.

Singleton elimination A new feature of this version is the possibility to do an informative elimination a non informative singleton definition. A *singleton definition* has only one constructor and all the argument of this constructor are non informative. In that case, there is a canonical way to interpret the informative extraction on an object in that type, such that the elimination on sort s is legal. Typical examples are the conjunction of non-informative propositions and the equality. In that case, the term `eq_rec` which was defined as an axiom, is now a term of the calculus.

```
Coq < Print eq_rec.

Coq < Extraction eq_rec.
```

Type of branches. Let c be a term of type C, we assume C is a type of constructor for an inductive definition I. Let P be a term that represents the property to be proved. We assume r is the number of parameters.

We define a new type $\{c:C\}^P$ which represents the type of the branch corresponding to the $c:C$ constructor.

$$\begin{array}{ll} \{c:(I_i\ p_1\ldots p_r\ t_1\ldots t_p)\}^P & \equiv (P\ t_1\ldots\ t_p c) \\ \{c:(x:T)C\}^P & \equiv (x:T)\{(c\ x):C\}^P \end{array}$$

We write $\{c\}^P$ for $\{c:C\}^P$ with C the type of c.

Examples. For list_A the type of P will be $\mathsf{list}_A \rightarrow s$ for $s \in \{\mathsf{Prop},\mathsf{Set},\mathsf{Type}(i),\mathsf{Typeset}(i)\}$.
$\{(\mathsf{cons}\ A)\}^P \equiv (a:A)(l:\mathsf{list}_A)(P\ (\mathsf{cons}\ A\ a\ l))$.

For Length_A, the type of P will be $(l:\mathsf{list}_A)(n:\mathsf{nat})(\mathsf{Length}_A\ l\ n) \rightarrow \mathsf{Prop}$ and the expression $\{(\mathsf{Lcons}\ A)\}^P$ is defined as :
$(a:A)(l:\mathsf{list}_A)(n:\mathsf{nat})(h:(\mathsf{Length}_A\ l\ n))(P\ (\mathsf{cons}\ A\ a\ l)\ (\mathsf{S}\ n)\ (\mathsf{Lcons}\ A\ a\ l\ n\ l))$.
If P does not depend on its third argument, we find the more natural expression :
$(a:A)(l:\mathsf{list}_A)(n:\mathsf{nat})(\mathsf{Length}_A\ l\ n) \rightarrow (P\ (\mathsf{cons}\ A\ a\ l)\ (\mathsf{S}\ n))$.

Typing rule. Our very general destructor for inductive definition enjoys the following typing rule :

$$\textbf{Case}\quad \frac{E[\Gamma] \vdash c : (I\ q_1 \ldots q_r\ t_1 \ldots t_s) \quad E[\Gamma] \vdash P : B \quad [(I\ q_1 \ldots q_r)|B] \quad (E[\Gamma] \vdash f_i : \{(c_{p_i}\ q_1 \ldots q_r)\}^P)_{i=1\ldots l}}{E[\Gamma] \vdash <\!P\!>\mathsf{Case}\ c\ \mathsf{of}\ f_1 \ldots f_l\ \mathsf{end} : (P\ t_1 \ldots t_s\ c)}$$

provided I is an inductive type in a declaration $\mathsf{Ind}(\Delta)[\Gamma_P](\ \Gamma_I := \Gamma_C\ )$ with $|\Gamma_P| = r$, $\Gamma_C = [c_1 : C_1; \ldots ; c_n : A_n]$ and $c_{p_1} \ldots c_{p_l}$ are the only constructors of I.

Example. For list and Length the typing rules for the `Case` expression are (writing just $t : M$ instead of $E[\Gamma] \vdash t : M$, the environment and context being the same in all the judgments).

$$\frac{l : \mathsf{list}_A \quad P : \mathsf{list}_A \to s \quad f_1 : (P\ (\mathsf{nil}\ A)) \quad f_2 : (a : A)(l : \mathsf{list}_A)(P\ (\mathsf{cons}\ A\ a\ l))}{<\!P\!>\mathsf{Case}\ l\ \mathsf{of}\ f_1\ f_2\ \mathsf{end} : (P\ c)}$$

$$\frac{\begin{array}{c} H : (\mathsf{Length}_A\ L\ N) \\ P : (l : \mathsf{list}_A)(n : \mathsf{nat})(\mathsf{Length}_A\ l\ n) \to \mathsf{Prop} \\ f_1 : (P\ (\mathsf{nil}\ A)\ \mathsf{O}\ \mathsf{Lnil}) \\ f_2 : (a : A)(l : \mathsf{list}_A)(n : \mathsf{nat})(h : (\mathsf{Length}_A\ l\ n))(P\ (\mathsf{cons}\ A\ a\ n)\ (\mathsf{S}\ n)\ (\mathsf{Lcons}\ A\ a\ l\ n\ h)) \end{array}}{<\!P\!>\mathsf{Case}\ H\ \mathsf{of}\ f_1\ f_2\ \mathsf{end} : (P\ L\ N\ H)}$$

Definition of ι-reduction. We still have to define the ι-reduction in the general case.

A ι-redex is a term of the following form :

$$<\!P\!>\mathsf{Case}\ (c_i\ p_1 \ldots p_r\ a_1 \ldots a_m)\ \mathsf{of}\ f_1 \ldots f_l\ \mathsf{end}$$

with c_i the i-th constructor of the inductive type I with r parameters.

The ι-contraction of this term is $(f_i\ a_1 \ldots a_m)$ leading to the general reduction rule :

$$<\!P\!>\mathsf{Case}\ (c_i\ p_1 \ldots p_r\ a_1 \ldots a_m)\ \mathsf{of}\ f_1 \ldots f_n\ \mathsf{end} \triangleright_\iota (f_i\ a_1 \ldots a_m)$$

6.5.5 Fixpoint definitions

The second operator for elimination is fixpoint definition. This fixpoint may involve several mutually recursive definitions. The basic syntax for a recursive set of declarations is

$$\mathsf{Fix}\ \{f_1 : A_1 := t_1 \ldots f_n : A_n := t_n\}$$

The terms are obtained by projections from this set of declarations and are written $\mathsf{Fix}\ f_i\{f_1 : A_1 := t_1 \ldots f_n : A_n := t_n\}$

Typing rule

The typing rule is the expected one for a fixpoint.

$$\textbf{Fix}\quad \frac{(E[\Gamma] \vdash A_i : s_i)_{i=1\ldots n} \quad (E[\Gamma, f_1 : A_1, \ldots, f_n : A_n] \vdash t_i : A_i :)_{i=1\ldots n}}{E[\Gamma] \vdash \mathsf{Fix}\ f_i\{f_1 : A_1 := t_1 \ldots f_n : A_n := t_n\} : A_i}$$

Any fixpoint definition cannot be accepted because non-normalizing terms will lead to proofs of absurdity.

The basic scheme of recursion that should be allowed is the one needed for defining primitive recursive functionals. In that case the fixpoint enjoys special syntactic restriction, namely one of the arguments belongs to an inductive type, the function starts with a case analysis and recursive calls are done on variables coming from patterns and representing subterms.

For instance in the case of natural numbers, a proof of the induction principle of type

$$(P:\mathsf{nat}\rightarrow\mathsf{Prop})(P\ \mathsf{O})\rightarrow((n:\mathsf{nat})(P\ n)\rightarrow(P\ (\mathsf{S}\ n)))\rightarrow(n:\mathsf{nat})(P\ n)$$

can be represented by the term:

$$\begin{array}{l}[P:\mathsf{nat}\rightarrow\mathsf{Prop}][f:(P\ \mathsf{O})][g:(n:\mathsf{nat})(P\ n)\rightarrow(P\ (\mathsf{S}\ n))]\\ \mathsf{Fix}\ h\{h:(n:\mathsf{nat})(P\ n):=[n:\mathsf{nat}]{<}P{>}\mathsf{Case}\ n\ \mathsf{of}\ f\ [p:\mathsf{nat}](g\ p\ (h\ p))\ \mathsf{end}\}\end{array}$$

Before accepting a fixpoint definition as being correctly typed, we check that the definition is “guarded”. A precise analysis of this notion can be found in [?].

The first stage is to precise on which argument the fixpoint will be decreasing. The type of this argument should be an inductive definition.

For doing this the syntax of fixpoints is extended and becomes

$$\mathsf{Fix}\ f_i\{f_1/k_1:A_1:=t_1\dots f_n/k_n:A_n:=t_n\}$$

where k_i are positive integers. Each A_i should be a type (reducible to a term) starting with at least k_i products $(y_1:B_1)\dots(y_{k_i}:B_{k_i})A'_i$ and B_{k_i} being an inductive type.

Now in the definition t_i, if f_j occurs then it should be applied to at least k_j arguments ant the k_j-th argument should be syntactically recognized as structurally smaller than y_{k_i}

The definition of being structurally smaller is a bit technical.

The main rules are the following:

Given a variable y of type an inductive definition in a declaration $\mathsf{Ind}(\Gamma)[\Gamma_P](\ \Gamma_I:=\Gamma_C\ )$ where Γ_I is $[I_1:A_1;\dots;I_k:A_k]$, and Γ_C is $[c_1:C_1;\dots;c_n:C_n]$. The terms structurally smaller than y are :

- $(t\ u),[x:u]t$ when t is structurally smaller than y .
- ${<}P{>}\mathsf{Case}\ c\ \mathsf{of}\ f_1\dots f_n\ \mathsf{end}$ when each f_i is structurally smaller than y.
  If c is y or is structurally smaller than y, its type is an inductive definition I_p part of the inductive declaration corresponding to y. Each f_i corresponds to a type of constructor $C_q\equiv(y_1:B_1)\dots(y_k:B_k)(I\ a_1\dots a_k)$ and can consequently be written $[y_1:B'_1]\dots[y_k:B'_k]g_i$. ($B'_i$ is obtained from B_i by substituting parameters variables) the variables y_j occurring in g_i corresponding to recursive arguments B_i (the ones in which one of the I_l occurs) are structurally smaller than y.

The following definitions are correct, we enter them using the `Fixpoint` command as described in section 2.5.4 and show the internal representation.

```
Coq < Fixpoint plus [n:nat] : nat -> nat :=
Coq <  [m:nat]<nat>Case n of m [p:nat](S (plus p m)) end.

Coq < Print plus.
```

```
Coq < Fixpoint lgth [A:Set;l:(list A)] : nat :=
Coq <   <nat>Case l of O [a:A][l':(list A)](S (lgth A l')) end.
```

```
Coq < Print lgth.
```

```
Coq < Fixpoint sizet [t:tree] : nat
Coq <   := <nat>Case t of [f:forest](S (sizef f)) end
Coq < with     sizef [f:forest] : nat
Coq <   := <nat>Case f of O [t:tree][f:forest](plus (sizet t) (sizef f)) end.
```

```
Coq < Print sizet.
```

Reduction rule

Let F be the set of declarations : $f_1/k_1 : A_1 := t_1 \ldots f_n/k_n : A_n := t_n$. The reduction for fixpoints is :

$$(\mathsf{Fix}\ f_i\{F\}\ a_1 \ldots a_{k_i}) \triangleright_\iota t_i\{(f_k/\mathsf{Fix}\ f_k\{F\})_{k=1\ldots n}\}$$

when a_{k_i} starts with a constructor. This last restriction is needed in order to keep strong normalization and corresponds to the reduction for primitive recursive operators.

We can illustrate this behavior on examples.

```
Coq < Goal (n,m:nat)(plus (S n) m)=(S (plus n m)).
```

```
Coq < Reflexivity.
```

```
Coq < Abort.
```

```
Coq < Goal (f:forest)(sizet (node f))=(S (sizef f)).
```

```
Coq < Reflexivity.
```

```
Coq < Abort.
```

But assuming the definition of a son function from tree to forest:

```
Coq <   Definition sont : tree -> forest := [t]<forest>Case t of [f]f end.
```

The following is not a conversion but can be proved after a case analysis.

```
Coq < Goal (t:tree)(sizet t)=(S (sizef (sont t))).
```

```
Coq < (* this one fails *)
Coq < Reflexivity.
```

```
Coq < Destruct t.
```

```
Coq < Reflexivity.
```

The `Match ...with ...end` expression

A unary `Match...with ...end`. The `Match` operator which was a primitive notion in older presentations of the Calculus of Inductive definitions is now just a macro definition which generates the good combination of `Case` and `Fix` operators in order to generate an operator for primitive recursive definitions. It always considers an inductive definition as a single inductive definition.

The following examples illustrates this feature.

```
Coq < Definition nat_pr : (C:Set)C->(nat->C->C)->nat->C
Coq <   :=[C,x,g,n]<C>Match n with x g end.

Coq < Print nat_pr.


Coq < Definition forest_pr
Coq <   : (P:forest->Set)(P emptyf)->((t:tree)(f:forest)(P f)->(P (consf t f)))
Coq <     ->(f:forest)(P f)
Coq <   := [C,x,g,n]<C>Match n with x g end.
```

The principles of mutual induction can be automatically generated using the `Scheme` command described in section 8.5.

6.6 Coinductive types

The implementation contains also coinductive definitions, which are types inhabited by infinite objects. For more information see the additional documentation referenced in section 15.5.

Chapter 7

Theories Library

A number of libraries, containing various developments of Coq axiomatizations, is available in the theories directory. This is further structured into a basic library INIT of elementary logical and mathematical notions, various specific libraries for sets, lists, arithmetic, algebra. A large area contrib (at the same level than theories) of user-contributed contains more specific libraries. This chapter briefly reviews these developments.

7.1 INIT

This area concerns the basic axiomatizations which are available in the standard Coq system, plus a few optional ones. The standard ones, which are loaded when the system is built, in order to initialize the global context, are the ones listed in the Prelude module: Logic, Datatypes, Specif, Peano, and Wf. The optional ones, which are needed in various circumstances, comprise: Logic_Type, and Classical.

7.1.1 Logic

The Logic module starts with the definition of the standard (intuitionistic) logical connectives, explained as inductive constructions. Their usual infix syntax can be found in the module Logic-Syntax.

Propositional Connectives

First, we find propositional calculus connectives:

```
Coq < Inductive True : Prop := I : True.

Coq < Inductive False : Prop := .

Coq < Definition not := [A:Prop] A->False.

Coq < Inductive and [A,B:Prop] : Prop := conj : A -> B -> A/\B.

Coq < Section Projections.

Coq < Variables A,B : Prop.

Coq < Theorem proj1 : A/\B -> A.
```

```
Coq < Theorem proj2 : A/\B -> B.


Coq < End Projections.

Coq < Inductive or [A,B:Prop] : Prop
Coq <     := or_introl : A -> A\/B
Coq <      | or_intror : B -> A\/B.

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

Coq < Definition IF := [P,Q,R:Prop] (P/\Q) \/ (~P/\R).

Coq < Hint I conj or_introl or_intror.
```

Quantifiers

Then we find first-order quantifiers:

```
Coq < Definition all := [A:Set][P:A->Prop](x:A)(P x).

Coq < Syntactic Definition All := (all ?).

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

Coq < Syntactic Definition Ex := (ex ?).

Coq < Inductive ex2 [A:Set;P,Q:A->Prop] : Prop
Coq <     := ex_intro2 : (x:A)(P x)->(Q x)->(ex2 A P Q).

Coq < Syntactic Definition Ex2 := (ex2 ?).
```

Equality

Then, we find equality, defined as an inductive relation. That is, given a `Set A` and an `x` of type `A`, the predicate `(eq A x)` is the smallest which contains `x`. This definition, due to Christine Paulin-Mohring, is equivalent to define `eq` as the smallest reflexive relation, and it is also equivalent to Leibniz' equality.

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

Coq < Hint refl_equal.
```

It is possible to write `x=y` for `(eq ? x y)`. The type of the arguments `x` and `y` is automatically synthesized (look at the `LogicSyntax.v` file, for more details).

Lemmas

Finally, a few easy lemmas are provided.

```
Coq < Theorem absurd : (A:Prop)(C:Prop) A -> ~A -> C.
```

```
Coq < Section equality.
Coq <  Variable A,B : Set.
Coq <  Variable f   : A->B.
Coq <  Variable x,y,z : A.
Coq <  Theorem sym_equal : x=y -> y=x.

Coq <  Theorem trans_equal : x=y -> y=z -> x=z.

Coq <  Theorem f_equal : x=y -> (f x)=(f y).

Coq <  Theorem sym_not_equal : ~(x=y) -> ~(y=x).

Coq < End equality.
Coq < Immediate sym_equal sym_not_equal.
```

7.1.2 Datatypes

Next, we find the definition of the basic data-types of programming, again defined as inductive constructions over the sort `Set`.

Programming

```
Coq < Inductive unit : Set := tt : unit.

Coq < Inductive bool : Set := true : bool
Coq <                       | false : bool.

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

Note that zero is the letter `O`, and *not* the numeral `0`.

We then define the disjoint sum of `A+B` of two sets `A` and `B`, and their product `A*B`.

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

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

```
Coq < Section projections.
Coq <    Variables A,B:Set.
Coq <    Definition fst := [H:A*B]<A> Case H of [x:A][y:B]x end.
Coq <    Definition snd := [H:A*B]<B> Case H of [x:A][y:B]y end.
Coq < End projections.
Coq < Syntactic Definition Fst := (fst ? ?).
Coq < Syntactic Definition Snd := (snd ? ?).
Coq < Hint pair inl inr.
```

7.1.3 Specif

The `Specif` module concerns notions about Sets that contain logical information. The usual infix syntax can be found in the module `SpecifSyntax`.

For instance, given `A:Set` and `P:A->Prop`, the construct `{x:A | (P x)}` (in abstract syntax `(sig A P)`) is a `Set`. We may build elements of this set as `(exist x p)` whenever we have a witness `x:A` with its justification `p:(P x)`.

From such a `(exist x p)` we may in turn extract its witness `x:A` (using an elimination construct such as `Case`) but *not* its justification, which stays hidden, like in an abstract data type. In technical terms, one says that `sig` is a "weak (dependent) sum". A variant `sig2` with two predicates is also provided.

```
Coq < Inductive  sig [A:Set;P:A->Prop] : Set
Coq <      := exist : (x:A)(P x) -> (sig A P).
Coq < Inductive sig2 [A:Set;P,Q:A->Prop] : Set
Coq <      := exist2 : (x:A)(P x) -> (Q x) -> (sig2 A P Q).
```

A "strong (dependent) sum" `{x:A & (P x)}` may be also defined, when the predicate `P` is now defined as a `Set` constructor.

```
Coq < Inductive sigS [A:Set;P:A->Set] : Set
Coq <      := existS : (x:A)(P x) -> (sigS A P).
Coq < Section projections.
Coq <    Variable A:Set.
Coq <    Variable P:A->Set.
Coq <    Definition projS1 := [H:(sigS A P)]<A> Case H of [x:A][h:(P x)]x end.
Coq <    Definition projS2 := [H:(sigS A P)]<[H:(sigS A P)](P (projS1 H))>
Coq <                              Case H of [x:A][h:(P x)]h end.
Coq < End projections.
Coq < Inductive sigS2 [A:Set;P,Q:A->Set] : Set
Coq <      := existS2 : (x:A)(P x) -> (Q x) -> (sigS2 A P Q).
```

A related non-dependent construct is the constructive sum {A}+{B} of two propositions A and B.

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

This sumbool construct may be used as a kind of indexed boolean data type. An intermediate between sumbool and sum is the mixed sumor which combines A:Set and B:Prop in the Set A+{B}.

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

We may define variants of the axiom of choice, like in Martin-Löf's Intuitionistic Type Theory.

```
Coq < Lemma Choice : (S,S':Set)(R:S->S'->Prop)((x:S){y:S'|(R x y)})
Coq <                        -> {f:S->S'|(z:S)(R z (f z))}.
```

```
Coq < Lemma Choice2 : (S,S':Set)(R:S->S'->Set)((x:S){y:S' & (R x y)})
Coq <                        -> {f:S->S' & (z:S)(R z (f z))}.
```

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

The next construct builds a sum between a data type A:Set and an exceptional value encoding errors:

```
Coq < Inductive Exc [A:Set] : Set := value : A->(Exc A)
Coq <                               | error : (Exc A).
```

This module ends with two axioms, relating the sorts Set and Prop in a way which is consistent with the realizability interpretation.

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

7.1.4 Peano

This module gives a few elementary properties of natural numbers, together with the definitions of predecessor, addition and multiplication.

```
Coq < Theorem eq_S : (n,m:nat) n=m -> (S n)=(S m).
```

```
Coq < Definition pred : nat->nat
Coq <       := [n:nat](<nat>Case n of (* O *) O
Coq <                             (* S u *) [u:nat]u end).
```

```
Coq < Theorem pred_Sn : (m:nat) m=(pred (S m)).
```

```
Coq < Theorem eq_add_S : (n,m:nat) (S n)=(S m) -> n=m.
```

```
Coq < Immediate eq_add_S.
```

```
Coq < Theorem not_eq_S : (n,m:nat) ~(n=m) -> ~((S n)=(S m)).
```

```
Coq < Hint not_eq_S.
```

```
Coq < Definition IsSucc : nat->Prop
Coq <    := [n:nat](<Prop>Case n of (* O *) False
Coq <                           (* S p *) [p:nat]True end).
```

```
Coq < Theorem O_S : (n:nat) ~(O=(S n)).
```

```
Coq < Theorem n_Sn : (n:nat) ~(n=(S n)).
```

```
Coq < Fixpoint plus [n:nat] : nat -> nat :=
Coq <     [m:nat](<nat>Case n of
Coq <        (* O *) m
Coq <      (* S p *) [p:nat](S (plus p m)) end).
```

```
Coq < Lemma plus_n_O : (n:nat) n=(plus n O).
```

```
Coq < Hint plus_n_O.
```

```
Coq < Lemma plus_n_Sm : (n,m:nat) (S (plus n m))=(plus n (S m)).
```

```
Coq < Hint plus_n_Sm.
```

```
Coq < Fixpoint mult [n:nat] : nat -> nat :=
Coq <     [m:nat](<nat> Case n of (* O *) O
Coq <                            (* S p *) [p:nat](plus m (mult p m)) end).
```

```
Coq < Lemma mult_n_O : (n:nat) O=(mult n O).
```

```
Coq < Hint mult_n_O.

Coq < Lemma mult_n_Sm : (n,m:nat) (plus (mult n m) n)=(mult n (S m)).


Coq < Hint mult_n_Sm.
```

Finally, it gives the definition of the usual orderings `le`, `lt`, `ge`, and `gt`.

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

Coq < Hint le_n le_S.

Coq < Definition lt := [n,m:nat](le (S n) m).

Coq < Hint Unfold lt.

Coq < Definition ge := [n,m:nat](le m n).

Coq < Hint Unfold ge.

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

Coq < Hint Unfold gt.
```

Properties of these relations are not initially known, but may be required by the user from modules `Le` and `Lt`. Finally, `Peano` gives some lemmas allowing pattern-matching, and a double induction principle.

```
Coq < Theorem nat_case : (n:nat)(P:nat->Prop)(P O)->((m:nat)(P (S m)))->(P n).


Coq < Theorem nat_double_ind : (R:nat->nat->Prop)
Coq <      ((n:nat)(R O n)) -> ((n:nat)(R (S n) O))
Coq <      -> ((n,m:nat)(R n m)->(R (S n) (S m)))
Coq <      -> (n,m:nat)(R n m).
```

7.1.5 Wf

The `Wf` module contains the basics of well-founded induction.

```
Coq < Chapter Well_founded.

Coq < Variable A : Set.

Coq < Variable R : A -> A -> Prop.

Coq < Inductive Acc : A -> Prop
Coq <    := Acc_intro : (x:A)((y:A)(R y x)->(Acc y))->(Acc x).

Coq < Lemma Acc_inv : (x:A)(Acc x) -> (y:A)(R y x) -> (Acc y).
```

```
Coq < Transparent Acc_inv.

Coq < Section AccRec.

Coq < Variable P : A -> Set.

Coq < Variable F : (x:A)((y:A)(R y x)->(Acc y))->((y:A)(R y x)->(P y))->(P x).

Coq < Fixpoint Acc_rec [x:A;a:(Acc x)] : (P x)
Coq <    := (F x (Acc_inv x a) ([y:A][h:(R y x)](Acc_rec y (Acc_inv x a y h)))).

Coq < End AccRec.

Coq < Definition well_founded := (a:A)(Acc a).

Coq < Theorem well_founded_induction :
Coq <     well_founded ->
Coq <         (P:A->Set)((x:A)((y:A)(R y x)->(P y))->(P x))->(a:A)(P a).


Coq < End Well_founded.
```

7.2 Other general-purpose libraries

7.2.1 Logic_Type

This module contains the definition of logical quantifiers axiomatized at the `Type` level.

```
Coq < Definition allT := [A:Type][P:A->Prop](x:A)(P x).

Coq < Syntactic Definition AllT := (allT ?).

Coq < Section universal_quantification.

Coq < Variable A : Type.

Coq < Variable P : A->Prop.

Coq < Theorem inst :  (x:A)(AllT P)->(P x).


Coq < Theorem gen : (B:Prop)(f:(y:A)B->(P y))B->(AllT P).


Coq < End universal_quantification.

Coq < Inductive  exT [A:Type;P:A->Prop] : Prop
Coq <     := exT_intro : (x:A)(P x)->(exT A P).

Coq < Syntactic Definition ExT := (exT ?).

Coq < Inductive exT2 [A:Type;P,Q:A->Prop] : Prop
Coq <     := exT_intro2 : (x:A)(P x)->(Q x)->(exT2 A P Q).

Coq < Syntactic Definition ExT2 := (exT2 ?).
```

Finally, it defines Leibniz equality `x==y` when `x` and `y` belong to `A:Type`.

```
Coq < Inductive eqT [A:Type;x:A] : A -> Prop
Coq <                         := refl_eqT : (eqT A x x).

Coq < Hint refl_eqT.

Coq < Section Equality_is_a_congruence.

Coq < Variables A,B : Type.

Coq < Variable  f : A->B.

Coq <  Variable x,y,z : A.

Coq <  Lemma sym_eqT : (x==y) -> (y==x).

Coq <  Lemma trans_eqT : (x==y) -> (y==z) -> (x==z).

Coq <  Lemma congr_eqT : (x==y)->((f x)==(f y)).

Coq < End Equality_is_a_congruence.

Coq < Immediate sym_eqT.

Coq < Inductive eqTS [A:Typeset;x:A] : A -> Prop
Coq <                         := refl_eqTS : (eqTS A x x).
```

It is possible to write `x==y` for `(eqT ? x y)`. The type of the arguments `x` and `y` is automatically synthesized (look at the `Logic_TypeSyntax.v` file, for more details).

7.2.2 Classical

The module `Classical` contains the rudiments of classical reasoning, starting with the excluded middle axiom, and with various versions of de Morgan's laws.

Beware. `Classical` is not provided in the default initialization of the system, you must require it explicitly (with `Require Classical`) if you need classical reasoning.

```
Coq < Axiom classic: (P:Prop)(P \/ ~(P)).

Coq < Lemma NNPP : (p:Prop)~(~(p))->p.

Coq < Lemma not_all_ex_not : (P:nat->Prop)(~((n:nat)(P n)) ->
Coq <                              (Ex [n:nat]~(P n))).

Coq < Lemma not_ex_all_not : (P:nat->Prop) ~(Ex [n:nat](P n)) -> (n:nat)~(P n).

Coq < Lemma ex_not_not_all : (P:nat->Prop) (Ex [n:nat]~(P n)) -> ~(n:nat)(P n).

Coq < Lemma all_not_not_ex : (P:nat->Prop) ((n:nat)~(P n)) -> ~(Ex [n:nat](P n)).
```

7.3 User contributions

Numerous user contributions are provided in the directory `contrib`. If you wish to add a contribution to Coqś library, write to `Gerard.Huet@inria.fr`.

Chapter 8

Tactics for inductive types and families

This chapter details a few special tactics useful for inferring facts from inductive hypotheses. They can be considered as tools that macro-generate complicated uses of the basic elimination tactics for inductive types.

The first section presents inversion tactics and the second describes a command `Scheme` for automatic generation of induction schemes for mutual inductive types.

8.1 Generalities about inversion

When working with inductive predicates, we are very often faced to some of these situations :

- we have an inconsistent instance of an inductive predicate in the local context of hypotheses. Thus, the current goal can be trivially proved by absurdity. This situation arises frequently when reasoning by induction on several arguments of an inductive predicate.
- we have an hypothesis that is an instance of an inductive predicate, and the instance has some variables whose constraints we would like to derive.

The inversion tactics are very useful to simplify the work in these cases. Inversion tools can be classified in three groups :

1. tools for inverting an instance without stocking the inversion lemma in the context : `Simple Inversion`, `Inversion` and `Inversion_clear`.
2. tools for generating and stocking in the context the inversion lemma corresponding to an instance : `Derive Inversion`, `Derive Inversion_clear`.
3. tools for inverting an instance using an already defined inversion lemma : `Use Inversion`.

As inversion proofs may be large in size, we recommend the user to stock the lemmas whenever the same instance needs to be inverted several times.

Let's consider, for example purposes, the type `listnat` of lists of natural numbers :

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

and the predicate `concat` such that `(concat x y z)` holds only when `z` is the list concatenation of `x` and `y` :

```
Coq < Inductive  concat : listnat->listnat ->listnat->Prop :=
Coq <   conc_nil : (x:listnat) (concat nil x x)
Coq < | conc_cons : (x,y,z:listnat) (n:nat)
Coq <                   (concat x y z) -> (concat (cons n x) y (cons n z)).
```

In the examples, we will use the following variables :

```
Coq < Variable P,Q,R: nat -> Prop.
```

8.2 Inverting an instance

- `Simple Inversion` *namehyp*

  Let *namehyp* be an hypothesis of type $(I\ \vec{t})$ in the local context. Let I is an inductive predicate. Then, `Simple Inversion` applied to *namehyp* derives **for each** constructor c_i of I, **all** the necessary constraints that should hold for the instance $(I\ \vec{t})$ to be proved by c_i.

  Suppose we have the following goal :

```
Coq < Show.
```

  The inductive type `concat` defines the smallest set closed by the constructors `conc_nil` and `conc_cons`. That means, that all instances of this type should be constructed by composition of these operators.

  Thus, from `H` we would deduce `n=m` and `(concat x y z)`, for the only constructor that allows to prove this kind of instance is `conc_cons`. More generally, given a certain instance Y of an inductive type, we can inspect all the constructors of the type, to determine for each one, the set of constraints that should be verified for Y to be proved with this constructor. This is commonly known as inverting the predicate. `Simple Inversion` performs this task.

  `Simple Inversion` applied to `H` yields two subgoals. The first one will contain in its context, all the constraints that should hold for `(concat (cons n x) y (cons m z))` to be constructed by `conc_nil`. Obviously, some of them will be inconsistent and will allow to prove the goal just by discrimination. The context of the second subgoal contains the informative constraints corresponding to the constructor `conc_cons` :

```
Coq < Simple Inversion H.
```

```
Coq < Show 2.
```

  As we said, the first subgoal can be proved easily by discriminating on `H0`. The second one has the interesting constraints : from `H1` we can deduce (by injection) that `n0=n` and `x0=x`, analogously, from `H3` we derive `n0=m` and `z0=z`. Then it is not difficult to derive that `(concat x y z)`.

- `Inversion` *namehyp*
  If the type of *namehyp* in the local context is $(I\ \vec{t})$, where I is an inductive predicate then, this tactic behaves as `Simple Inversion` deriving all the constraints and making the following simplifications :

  – if among the derived constraints, there is one that is a discriminable equality, then it proves the branch automatically by discrimination.

  – if no discriminable equality was derived, it applies all possible injections in order to obtain rules for rewriting in the conclusion.

```
Coq < Undo.
```

```
Coq < Inversion H.
```

The branch corresponding to `conc_nil` was automatically eliminated. The goal to prove corresponds to the constructor `conc_cons`. Note that the hypotheses `(cons n0 x0)=(cons n x)` and `(cons n0 z0)=(cons m z)` had been simplified, and a substitution has been done on the goal using the constraint `n=m`.

- `Inversion_clear` *namehyp*
  This tactic behaves as `Inversion` but it erases the inverted hypothesis and the simplified equations from the local context.

```
Coq < Undo.
```

```
Coq < Inversion_clear H.
```

Now we have the goal substituted and the expected hypothesis `(concat x y z)` resulting from the inversion. Note that `H` has disappeared from the context : `Inversion_clear` always **replaces** the inverted hypothesis by the derived constraints.

Variants :

1. `Inversion` *namehyp* `in` $h_1 \dots h_n$
   Let $h_1 \dots h_n$, be identifiers in the local context. This tactic behaves as generalizing $h_1 \dots h_n$, and then performing `Inversion`.

2. `Inversion_clear` *namehyp* `in` $h_1 \dots h_n$
   Let $h_1 \dots h_n$, be identifiers in the local context. This tactic behaves as generalizing $h_1 \dots h_n$, and then performing `Inversion_clear`.

8.3 Deriving the inversion lemmas

The tactics `Inversion` and `Inversion_clear` work on a certain instance $(I\ \vec{t})$ of an inductive predicate. At each application, they inspect the given instance and derive the corresponding inversion lemma. If we have to inverse the same instance several times it is recommended to stock the lemma in the context and just reusing it whenever we need it.

The families of commands `Derive Inversion` and `Derive Inversion_clear` allow to generate the inversion lemma from a given instance. In next section we describe the tactic `Use Inversion` that refines the goal with a specified inversion lemma.

- `Derive Inversion` *name* `with` $(\vec{x}:\vec{T})(I\ \vec{t})$
  Let I be an inductive predicate. This command generates and stocks the inversion lemma corresponding to the instance $(\vec{x}:\vec{T})(I\ \vec{t})$ with the name *name* in the **global** environment. When applied it is be equivalent to having inverted the instance with the tactic `Inversion`.

- `Derive Inversion_clear` *name* `with` $(\vec{x}:\vec{T})(I\ \vec{t})$
  Let I be an inductive predicate. This command generates and stocks the inversion lemma corresponding to the instance $(\vec{x}:\vec{T})(I\ \vec{t})$ with the name *name* in the **global** environment. When applied it is be equivalent to having inverted the instance with the tactic `Inversion_clear`.

  For example, to invert the instance `(concat (cons n x) y (cons  m z))` we can do :

```
Coq < Derive Inversion_clear leminv1 with
Coq <      (n,m:nat)(x,y,z:listnat)(concat (cons n x) y (cons m z)).
```

We can inspect the generated lemma by `Check`, its type is :

```
Coq < Check leminv1.
```

The derived inversion lemmas are adequate for inverting the instance with which it was generated. The tactic `Derive` applied to different instances yields different lemmas. In general, if we invert an instance $(\vec{x}:\vec{T})(I\ \vec{t})$ the inversion lemma will expect a predicate of type $(\vec{x}:\vec{T})Prop$ as first argument. The reason is that `Derive` considers the global variables occurring in the instance as "constants".

For example, if we define :

```
Coq < Variable a,b:nat.

Coq < Variable l1,l2,l3:listnat.

Coq < Derive Inversion_clear leminv_var with (concat (cons a l1) l2 (cons b l3)).
```

The derived lemma is different to `leminv1` :

```
Coq < Check leminv_var.
```

Note that while `leminv_var` expects a proposition as argument, `leminv1` expects a predicate.

Variants :

1. `Derive Inversion` *name namehyp*
   Let *namehyp* have type $(I\ \vec{t})$ in the local context (I is an inductive predicate). Then, this command has the same semantics as `Derive Inversion` *name* `with` $(\vec{x}:\vec{T})(I\ \vec{t})$ where $\vec{x}$ are the free variables of $(I\ \vec{t})$ declared in the local context (variables of the global context are considered as constants).

2. `Derive Inversion_clear` *num name namehyp*
   Let *namehyp* have type $(I\ \vec{t})$ in the local context (I an inductive predicate). Then, this command has the same semantics as `Derive Inversion_clear` *name* `with` $(\vec{x} : \vec{T})(I\ \vec{t})$ where $\vec{x}$ are the free variables of $(I\ \vec{t})$ declared in the local context (variables of the global context are considered as constants).

   If we do this on our example:

```
Coq < Undo.
```

   We can stock the inversion lemma corresponding to `H` :

```
Coq < Derive Inversion_clear leminv2 H.
```

   The type of `leminv2` is the same as `leminv1`'s for both lemmas were built from the same instance :

3. `Derive Inversion` *num name namehyp*
   This command behaves as `Derive Inversion` *name namehyp* performed on the goal number *num*.

4. `Derive Inversion_clear` *num name namehyp*
   This command behaves as `Derive Inversion_clear` *name namehyp* performed on the goal number *num*.

8.4 Using already defined inversion lemmas

- `Use Inversion` *namehyp leminv*
  Let *namehyp* have type $(I\ \vec{t})$ (I an inductive predicate) in the local context, and *leminv* be an inversion lemma. Then, this tactic refines the current goal with this lemma.

  To invert `H` we can use either `leminv1` or `leminv2` :

```
Coq < Use Inversion H leminv2.
```

  In this case applying `leminv1` or `leminv2` is the same because both lemmas were built from the same instance and have the same type. This is not the case of `leminv_var`. Suppose we have the following goal :

```
Coq < Show.
```

  When we apply `leminv1`, the system will refine the current goal with this lemma. For that it instantiates the lemma with the predicate `[n,m:nat][x,y,z:listnat](P n)` of type `nat->nat->listnat->listnat->listnat->Prop` :

```
Coq < Use Inversion H leminv1.
```

  If we use `leminv_var` the result is different :

```
Coq < Undo.
```

```
Coq < Use Inversion H leminv_var.
```

No rewriting was done using the fact that `a=b`, the lemma was applied as expected to the proposition `(P a)`. Beware when using inversion lemmas generated from instances with global variables. In a certain sense the inversion we obtain is partial, as not necessarily all the constraints are neither explicit nor used.

Variants :

1. `Use Inversion` *namehyp leminv* `in` $h_1 \ldots h_n$
   This tactic has the same semantics as the previous one, but rewrites not only in the conclusion but **also** in the hypotheses $h_1 \ldots h_n$.

8.5 `Scheme ...`

The `Scheme` command is a higher-level tool for generating automatically (possibly mutual) induction principles for given types and sorts. Its syntax follows the schema :
`Scheme` $ident_1$ `:= Induction for` $term_1$ `Sort` $sort_1$
`with`
  ..
`with` $ident_m$ `:= Induction for` $term_m$ `Sort` $sort_m$
$term_1 \ldots term_m$ are different inductive types belonging to the same packages of mutual inductive definitions. This command generates $ident_1 \ldots ident_m$ to be mutually recursive definitions. Each term $ident_i$ proves a general principle of mutual induction for objects in type $term_i$.
Example : The definition of principle of mutual induction for `tree` and `forest` over the sort `Set` is defined by the command :

```
Coq < Scheme tree_forest_rec := Induction for tree Sort Set
Coq < with forest_tree_rec := Induction for forest Sort Set.
```

You may now look at the type of `tree_forest_rec` :

```
Coq < Check tree_forest_rec.
```

This principle involves two different predicates for `trees` and `forests` it also has three premisses each one corresponding to a constructor of one of the inductive definition.

The principle `tree_forest_rec` shares exactly the same premisses, only the conclusion now refers to the property of forests.

```
Coq < Check forest_tree_rec.
```

Variants : `Scheme` $ident_1$ `:= Minimality for` $term_1$ `Sort` $sort_1$
`with`
  ..
`with` $ident_m$ `:= Minimality for` $term_m$ `Sort` $sort_m$
Same as before but define a non-dependent elimination principle more natural in case of inductively defined relations.
Example : With the predicates `odd` and `even` inductively defined as :

```
Coq < Mutual Inductive odd : nat->Prop :=
Coq <      oddS : (n:nat)(even n)->(odd (S n))
Coq < with even : nat -> Prop :=
Coq <      evenO : (even O)
Coq <    | evenS : (n:nat)(odd n)->(even (S n)).
```

The command following command generates a powerful elimination principle :

```
Coq < Scheme odd_even := Minimality for odd Sort Prop
Coq < with   even_odd := Minimality for even Sort Prop.
```

The type of `odd_even` for instance will be :

```
Coq < Check odd_even.
```

The type of `even_odd` will share the same premisses but the conclusion will be `(n:nat)(even n)->(Q n)`.

Chapter 9

Syntax Extensions

9.1 Introduction

The Coq system allows us to define syntactic constants. It provides also a more sophisticated mechanism to extend the grammars of terms, vernacular commands and tactics.

Before starting the description of these features, we illustrate them by a detailed example. We use also some pretty-printing rules (see the pretty-printing manual).

Our example corresponds to a beginning of a development on functions.

The term `(explicit_comp A B C f g)` is the function obtained by composing the functions `f:A->B` and `g:B->C`.

```
Coq < Definition explicit_comp := [A,B,C:Set][f:A->B][g:B->C]
Coq <                              [a:A](g(f a)).
```

We define also the extensional equality on functions `explicit_ext`. Two functions `f` and `g` of domain `A` and codomain `B` are equal if their application at each element to `A` are equal (on `B`). We load the `Logic` module to be able to use the Leibniz equality "`=`".

```
Coq < Definition explicit_ext := [A,B:Set][f,g:A->B]
Coq <                             (x,y:A)(f x)=(g x).
```

With these definitions, we will try to prove the associativity of functional composition. We start by adding to the system the declarations we need.

```
Coq < Variable A, B, C, D : Set.

Coq <
Coq < Variable f : A -> B.

Coq <
Coq < Variable g : B -> C.

Coq <
Coq < Variable h : C -> D.
```

In the current version of Coq, it is no longer necessary to give explicitly all the arguments of an application. Indeed the system is able to synthesize some implicit arguments from the context. However we should indicate the place of the implicit arguments : they are represented by the "?" symbol. Thus, in our case we can write `(explicit_comp ? ? ? f g)` since the domain and codomain of `f` and `g` are implicitly given by their type. We will also write `(explicit_ext ? ? f g)`.

The statement of the associativity of functional composition becomes :

```
Coq < Lemma Ass :
Coq <   (explicit_ext ? ?
Coq <          (explicit_comp ? ? ? (explicit_comp ? ? ? f g) h)
Coq <            (explicit_comp ? ? ? f (explicit_comp ? ? ? g h))).
```

We will show two methods to improve this cumbersome syntax.

Using syntactic definitions :

Syntactic definitions are macros giving names to terms, possibly untyped. In our case, we give names to `(explicit_comp ? ? ?)` and `(explicit_ext ? ?)`, respectively `comp` and `ext`. Thus `(comp f g)` is equivalent to `(explicit_comp ? ? ? f g)`.

```
Coq < Syntactic Definition comp := (explicit_comp ? ? ?).
```

We do the same with `explicit_ext`.

```
Coq < Syntactic Definition ext := (explicit_ext ? ?).
```

We can now reformulate our goal.

```
Coq < Abort.

Coq <
Coq < Lemma Ass : (ext (comp (comp f g) h) (comp f (comp g h))).
```

Let us explain the result : during the parsing, the occurrences of `comp ext` are replaced by their syntactic definition. Then the system replaced each occurrence of "?" by a term it synthesized.

A problem subsists : the goal is printed with all the synthesized terms, and thus it seems obscure. The solution is to define pretty-printing rules that hide the synthesized terms. The following rule will print `(explicit_comp A B C f g)` as `(comp f g)`, ignoring the three first arguments. See the pretty-printing manual for the complete description of the `Syntax` command.

```
Coq < Syntax constr pp_comp <<(explicit_comp $A $B $C $f $g)>> 10
Coq <          "comp " <$f:"Term":L> " " <$g:"Term":L>.
```

We write also a pretty-printing rule for `explicit_ext` that prints `(explicit_ext A B f g)` as `(ext f g)`.

```
Coq < Syntax constr pp_ext <<(explicit_ext $A $B $f $g)>> 10
Coq <          "ext " <$f:"Term":L> " " <$g:"Term":L>.
```

To see the effect of these rules, we print the current goal.

```
Coq < Show.
```

Using grammar rules :

The previous solution is satisfactory, but is rather far from the usual mathematical notations. The extensible grammars will help us to improve our syntax.

For functional composition, we want to use an infix operator "o". Thus we enter a new grammar rule saying that the generic terms "`$f o $g`" have to be interpreted as `(explicit_comp ? ? ? $f $g)`. The non-terminals `command`, `command7` and `command8` are explained later.

```
Coq < Abort.
```

```
Coq <
Coq < Grammar command command8 :=
Coq <                [ command7($f) "o" command8($g) ] ->
Coq <                [$0=<<(explicit_comp ? ? ? $f $g)>>].
```

For `(explicit_ext ? ? f g)`, we choose the "`f =%ext g`" syntax since the "=" symbol is already used for the Leibniz equality and that overloading is not provided in this current version of Coq. Let us enter the corresponding grammar rule.

```
Coq < Token "%".
```

```
Coq < Grammar command command1 :=
Coq <                [ command0($f) "=" "%" "ext" command0($g) ] ->
Coq <                [$0=<<(explicit_ext ? ? $f $g)>>].
```

The associativity statement becomes :

```
Coq < Lemma Ass : ((f o g) o h) =%ext (f o (g o h)).
```

The `[Recompiling 1 nonterminal(s)...]` message indicates that the grammar was successfully extended, i.e. the two rules we add are not ambiguous.

We should change our pretty-printing rules in order to have a display in accordance with our syntax.

```
Coq < Syntax constr pp_comp <<(explicit_comp $A $B $C $f $g)>> 10
Coq <          <$f:"Term":L> " o " <$g:"Term":L>.
```

```
Coq <
Coq < Syntax constr pp_ext <<(explicit_ext $A $B $f $g)>> 1
Coq <          <$f:"Term":L> " =" "%" "ext " <$g:"Term":L>.
```

The old rules of `explicit_comp` and `explicit_ext` are overridden by those we enter.

We verify the result of our new pretty-printing rules.

```
Coq < Show.
```

We are now going to describe in details syntactic definitions and extensible grammars.

The associativity proof is left to the reader.

```
Coq < Abort.
```

9.2 Syntactic Definitions

The syntactic definitions define syntactic constants, i.e. give a name to a term possibly untyped but syntactically correct. Their syntax is :

`Syntactic Definition` *name* `:=` *term* `.`

Syntactic definitions behave like macros : every occurrence of a syntactic constant in an expression is immediately replaced by its body.

Let us extend our functional language with the definition of the identity function :

```
Coq < Definition explicit_id := [A:Set][a:A]a.
```

We declare also a syntactic definition `id` :

```
Coq < Syntactic Definition id := (explicit_id ?).
```

The term `(explicit_id ?)` is untyped since the implicit arguments cannot be synthesized. There is no type check during this definition. Let us see what happens when we use a syntactic constant in an expression like in the following example.

```
Coq < Eval (id O).
```

First the syntactic constant `id` is replaced by its body `(explicit_id ?)` in the expression. Then the resulting expression is evaluated by the typechecker, which fills in "?" place-holders.

The standard usage of syntactic definitions is to give names to terms applied to implicit arguments "?". In this case, a special command is provided :

`Syntactic Definition` *name* `:=` *term* `|` *n* `.`

The body of the syntactic constant is *term* applied to *n* place-holders "?".

We can define a new syntactic definition `id1` for `explicit_id` using this command. We changed the name of the syntactic constant in order to avoid a name conflict with `id`.

```
Coq < Syntactic Definition id1 := explicit_id | 1.
```

The new syntactic constant `id1` has the same behavior as `comp` :

```
Coq < Eval (id1 O).
```

Warnings :

- Syntactic constants defined inside a section are no longer available after closing the section.
- We cannot see the body of a syntactic constant with a `Print` command.

9.3 Extensible Grammars

The parsing process consists in reading an expression (a list of tokens) and deciding wether it belongs to the language or not. If it is, the parser transforms the expression into an internal form called AST (Abstract Syntax Tree). An expression belongs to the language if there exists a sequence of grammar rules that recognize it. The transformation to AST is performed by executing successively the *actions* bound to these rules. In Coq we can extend dynamically the language by adding new rules. We are going to describe this mechanism.

A grammar rule consists of :

- a grammar name : defined by a parser entry and a non-terminal. One can have two non-terminals of the same name if they are in different entries.
- a production : formed by a left member of production (LMP) and an action.

Let us comment the functional composition rule :

```
Grammar command command8 :=
            [ command7($f) "o" command8($g) ] ->
            [$0=<<(explicit_comp ? ? ? $f $g)>>].
```

The command above extends the grammar `command command8`, i.e. the grammar of entry `command` and of non-terminal `command8`. The new production is :

```
[ command7($f) "o" command8($g) ] -> [$0=<<(explicit_comp ? ? ? $f $g)>>]
```

`[ command7($f) "o" command8($g) ]` is the LMP and
`[$0=<<(explicit_comp ? ? ? $f $g)>>]` the action.

A grammar name can have parameters. They will be instantiated by ASTs during the application of the grammar production. Parameters are separated by ";" and enclosed between "[" and "]".

```
Coq < Grammar command command12[$p1;$p2] :=
Coq <               [ command0($p3) ] -> [$0=<<($p1=$p2)/\($p2=$p3)>>].
```

Grammars are dynamically extended by new productions as we need. A grammar name does not have to be explicitly defined : it is defined by giving its first production. All rules of a same grammar must have the same parameters. For instance, the following rule is refused because the `command command12` grammar has been already defined with two parameters.

```
Coq < Grammar command command12[$p1] := [ command5($c) ] -> [$0=$c].
```

A grammar may have several or zero productions. Assume that the `command command13` does not exist. The next command defines it with zero productions; of course, it may be extended later.

```
Coq < Grammar command command13 := .
```

9.3.1 Left Member of Productions (LMP)

A LMP is composed of a combination of tokens (enclosed between double quotes “"” and “"”) and grammar calls specifying the entry. It is enclosed between “[” and “]”.

The empty LMP, represented by `[ ]`, corresponds to ϵ in formal language theory.

A grammar call is done by $entry : nonterminal[a_1; \cdots ; a_n](\$id)$ where :

- *entry nonterminal* specifies the entry of the grammar, and the non-terminal.
- $a_1 \cdots a_n$ are actions, arguments of the called grammar. They must correspond to the number of parameters of the grammar *entry nonterminal*. Otherwise an error occurs with the message “`Bad number of arguments in the call of` *entry nonterminal*”. This verification is done during the use of the rule.
- $\$id$ is a metavariable that will receive the AST resulting from the call to the grammar.

The elements *entry* and ($\$id$) are optional. The grammar entry can be omitted if it is the same as the entry of the caller non-terminal. Also, ($\$id$) is omitted if we do not want to get back the AST result. Thus a grammar call can be reduced to a non-terminal.

When an LMP is used in the parsing process of an expression, it is analyzed from left to right. Every token met in the LMP should correspond to the current token of the expression. As to the grammars calls, they are performed in order to recognize parts of the initial expression.

For instance, let us see the behavior of the functional composition rules LMP.

```
Grammar command command8 :=
             [ command7($f) "o" command8($g) ] ->
             [$0=<<(explicit_comp ? ? ? $f $g)>>].
```

When this rule is selected, its LMP calls the grammar `command command7`. This grammar recognizes a term that it binds to the metavariable `$f`. Then it meets the token “`o`” and finally it calls the grammar `command command8`. This grammar returns the recognized term in `$g`. The function composition rule constructs the term `(explicit_comp ? ? ? $f $g)`.

Warning : Metavariables are identifiers preceded by the “`$`” symbol. They cannot be replaced by identifiers. For instance, if we enter the functional composition rule with identifiers and not metavariables, an error occurs.

```
Coq < Grammar command command8 := [ command7(f) "o" command8(g) ] ->
Coq <                              [$0=<<(explicit_comp ? ? ? f g)>>].
Error: ident found which was not a metavariable: f
```

9.3.2 Actions

Every rule should generate an AST corresponding to the syntactic construction that it recognizes. This generation is done by an *action*. Thus every rule is associated to an action.

As we have already seen in the previous examples, the LMP and the action are separated by “`->`”.

We distinguish two kinds of actions : the simple actions and the conditional actions.

Simple Actions

A simple action is presented as a list enclosed between "[" and "]" of bindings separated by ";". Each binding has the form *$id* = *pattern* where *$id* is a metavariable and *pattern* the description of an AST.

In each pattern may appear the metavariables defined at its left, i.e. parameters of the grammar, results of grammar calls and the metavariables of the previous bindings.
Among the bound metavariables must appear the metavariable `$0`. Indeed the AST generated by the action is the pattern bound to `$0`. All the bindings after the binding `$0` are ignored.
Example 1 : When an action should generate a big term, we can use intermediate bindings to construct it progressively. In the following example, from the syntax `t1*+t2` we generate the term `(plus (plus t1 t2) (mult t1 t2))`. For better clarity, we use the intermediate metavariables `$p1` and `$p2`.

```
Coq < Grammar command command1 :=
Coq <              [ command0($a) "*" "+" command0($b) ] ->
Coq <              [$p1=<<(plus $a $b)>>; $p2=<<(mult $a $b)>>;
Coq <               $0=<<(plus $p1 $p2)>>].
```

Let us give an example with this syntax :

```
Coq < Goal (O*+O)=O.
```

Example 2 : The rule below allows us to use the syntax `t1#t2` for the term `~t1=t2`.

```
Coq < Grammar command command1 :=
Coq <              [ command0($a) "#" command0($b) ] ->
Coq <              [$0=<<~($a=$b)>>].
```

For instance, let us give the statement of the symmetry of `#` :

```
Coq < Goal (A:Set)(a,b:A) a#b -> b#a.
```

Example 3 : We extend the `command command1` grammar with a rule that generates the term `t1=t2 /\ t2=t3` for the syntax `t1=t2=t3`.

```
Coq < Grammar command command1 :=
Coq <              [ command0($x) "=" command0($y) "="
Coq <                command12[[$0=$x];[$0=$y]]($r) ] -> [$0=$r].
```

During the parsing of `t1=t2=t3`, `t1` and `t2` are recognized by the grammars `command command0` and are respectively bound to `$x` and `$y`. Then we call `command command12` with the arguments `$x` and `$y`. We show its unique production.

```
Grammar command command12[$p1;$p2] := [ command0($p3) ] ->
                                       [$0=<<($p1=$p2)/\($p2=$p3)>>].
```

The parameters are instantiated by the arguments $x and $y, thus now $p1 and $p2 bind respectively the values of $x and $y, i.e. t1 and t2. The command command12 grammar recognizes t3 that it binds to $p3. Finally it generates the term t1=t2/\t2=t3 that is bound to $r. The result of the command command1 rule is also the value of $r.

As usual we check our new syntax on an example :

```
Coq < Goal (plus (S O) O)=(plus O (S O))=(S O).
```

Example 4 : Finally we give an example to show the importance of the order of bindings. In the example of *+, let us commute the last two bindings.

We change also the symbol *+ into +* to avoid ambiguity problems.

```
Coq < Grammar command command1 :=
Coq <             [ command0($a)  "+" "*" command0($b) ] ->
Coq <             [$p1=<<(plus $a $b)>>; $0=<<(plus $p1 $p2)>>;
Coq <              $p2=<<(mult $a $b)>>].
```

The parsing of O +* O fails because during the action interpretation, $p2 is used before it is bound.

```
Coq < Eval O +* O.
```

Example 5 : Let us see what happens if we enter in the system a rule with an action that does not bind $0.

```
Coq < Token "!".
```

```
Coq < Grammar command command1 :=
Coq <             [ command0($a) "!" "=" command0($b) ] ->
Coq <             [$1=<<$a#$b>>].
```

Nothing! The rule is accepted. Indeed, the verifications about the action of a rule are performed only during the use of this rule as you see in the following example :

```
Coq < Check O != O.
```

Conditional Actions

They are defined with the following syntax :

case $\$id$ of $pattern_1$ -> $action_1$ | $\cdots$ | $pattern_n$ -> $action_n$ esac

The action to execute is chosen according to the value of the metavariable $\$id$. This metavariable should be previously bound (for example, during a grammar call or as a parameter).

The matching is performed from left to right. The selected action is the one associated to the first pattern that matches the value of $\$id$. This matching operation will bind the metavariables appearing in the selected pattern.

Let us take an example. Suppose we want to change the syntax of dependent types. We enter a grammar rule that recognizes terms of the form |t1 in t2|t3 where t1, t2 and t3 are terms respectively recognizable by command lcommand, command command and command command grammars.

```
Coq < Grammar command command0 :=
Coq <              [ "|" lcommand($v) "in" command($type) "|"
Coq <                command($body) ] ->
Coq <              case $v of ($VAR $id) ->
Coq <                   [$0=<<($id:$type)$body>>] esac.
```

During the parsing of `|t1 in t2|t3` by this rule, the bindings (`$v`,`t1`), (`$type`,`t2`) and (`$body`,`t3`) are created. Then we compare the value of `$v`, i.e. `t1`, with the pattern `($VAR $id)` (representing the general form of a variable AST). If this matching succeeds, `$id` is bound to the identifier contained in `t1`.

We reformulate the statement of the symmetry of `#` :

```
Coq < Goal |a in nat||b in nat| a#b -> b#a.
```

In the case where the matching fails, i.e. no `case` pattern matchs the metavariable `$id`, the parsing fails and an error occurs. For instance :

```
Coq < Goal |(S O) in nat|O=O.
```

Our dependent type rule fails because `(S O)` is not a variable.

Several `case` structures can be interwoven since each $action_i$ can be also a `case` structure. Of course it should be finished by a simple action and the executed action will be the action finally selected.

Let us extend our previous example to recognize the dependent types `|t1,t2 in t3|t4`. We use two embedded `case` statements in order to verify that `t1` and `t2` are variables.

```
Coq < Grammar command command0 :=
Coq <              [ "|" lcommand($v1) "," lcommand($v2) "in"
Coq <                command($type) "|" command($body) ] ->
Coq <              case $v1 of ($VAR $id1) ->
Coq <                   case $v2 of ($VAR $id2) ->
Coq <                        [$0=<<($id1,$id2:$type)$body>>] esac
Coq <              esac.
```

We may use this syntax to write the symmetry of `#` in a more readable way :

```
Coq < Goal |a,b in nat| a#b -> b#a.
```

9.3.3 Entries

All the given examples concern the predefined entry `command`. However there exist other predefined entries. Each of them (except `prim`) possesses an initial grammar for starting the parsing process.

Four grammar entries are predefined.

- `command` : it is the term entry. It allows to have a pretty syntax for terms. Its initial grammar is `command command`. This entry contains several non-terminals, among them `command0` to `command10` which stratify the terms according to priority levels (0 to 10).

  Example : Let us see the grammar rules of conjunction and disjunction defined in the file `PreludeSyntax.v`. Conjunction is defined with the non-terminal `command6` and disjunction with `command7` : disjunction has a higher priority than conjunction. Thus `A/\B\/C` will be parsed as `A/\(B\/C)` and not as `(A/\B)\/C`. In the grammar rules, the character "\" must be doubled since it is the escape character of strings in Camllight, the implementation language of Coq.

  ```
  Grammar command command6 :=
               [ command5($c1) "/\\" command6($c2) ] ->
               [$0=<<(and $c1 $c2)>>].

  Grammar command command7 :=
               [ command6($c1) "\\/" command7($c2) ] ->
               [$0=<<(or $c1 $c2)>>].
  ```

  These priority levels allow us also to specify the order of associativity of operators. Thus conjunction and disjunction associate to the right since in both cases the priority of the right term (resp. `command6` and `command7`) is higher than the priority of the left term (resp. `command5` and `command6`).

- `vernac` : it is the vernacular command entry, with `vernac vernac` as initial grammar. Thanks to it, the developers can define the syntax of new commands they add to the system. As to users, they can change the syntax of the predefined vernacular commands.

- `tactic` : it is the tactic entry with `tactics tactic` as initial grammar. This entry allows to define the syntax of new tactics. See the tactics manual for more details.

- `prim` : it is the entry of the primitive grammars. The next section is devoted to it.

The user can define new entries.

```
Coq < Grammar newentry nonterm :=
Coq <              [ "&" command:command($c) ] -> [$0=$c].
```

The grammars of new entries do not have an initial grammar. To use them, they must be called (directly or indirectly) by grammars of predefined entries. We give an example of a (direct) call of the grammar `new-entry nonterm` by `command command`. This following rule allows to use the syntax `a&b` for the conjunction `a/\b`.

```
Coq < Grammar command command :=
Coq <              [ command8($a) newentry:nonterm($b) ] ->
Coq <              [$0=<<$a/\$b>>].
```

It is interesting to note that the basic syntax of the system is described by the extensible grammar mechanism. This syntax is described in the following files in the directory `src/syntax`.

- `Command.v`: term syntax.
- `Tactic.v`: vernacular command syntax.
- `Vernac.v`: tactic syntax.

To know the non-terminals in the predefined entries, on can consult these files.

9.3.4 Primitive Grammars

The primitive grammars are not defined by the extensible grammar mechanism. They are encoded inside the system.

The `prim` entry contains the following non-terminals :

- `ident` : identifier grammar.
- `number` : number grammar.
- `string` : string grammar.
- `unparsing` : pretty-printing grammar.
- `grammar_entry` : grammar of the extensible grammar mechanism. It corresponds to the non-terminal ⟨*Grammar_entry*⟩ in the figure 9.1.
- `spat` : pattern grammar.
- `raw_command` : AST grammar.

The primitive grammars are used as the other grammars; for instance the identifier grammar call is done by `prim:ident($id)`.

These primitive grammars cannot be extended. However the user can define new non-terminals in the `prim` entry, as for the other entries.

9.3.5 Patterns

Patterns describe AST to generate during the grammar rules application. They appear in the action part of grammar rules.
In the general case, the user does not have to put explicitly an AST in the action of his rules. Indeed, if the AST to generate corresponds to a well formed term, one can call a grammar to parse it and to return the AST result. For instance, in the functional composition grammar, the pattern bound to `$0` is `<<(explicit_comp ? ? ? $f $g)>>`.

Recall that this rule parses expressions of the form `t1 o t2` and generates the term `(explicit_comp ? ? ? t1 t2)`. This term is parsable by `command command` grammar. This grammar is invoked on this term to generate an AST by putting the term between "`<<`" and "`>>`".

We can also invoke the initial grammars of the other predefined entries.

- `<< t >>` parses `t` with `command command` grammar.
- `<:command:< t >>` parses `t` with `command command` grammar.
- `<:vernac:< t >>` parses `t` with `vernac vernac` grammar.
- `<:tactic:< t >>` parses `t` with `tactic tactic` grammar.

For a complete description of patterns and AST, see the pretty-printing manual.
Warning : We cannot invoke other grammars than those we described.

9.3.6 Other examples

We give some applications to the entries `vernac` and `tactic`.
Example 1 : Thanks to the following rule, "`|- term.`" will have the same effect as "`Goal term.`".

```
Coq < Token "-".

Coq < Grammar vernac vernac :=
Coq <                [ "|" "-" command:command($term) "." ] ->
Coq <                [$0=<:vernac:<Goal $term.>>].

Coq < |- (A:Prop)A->A.
```

Example 2 : We can adapt the vernacular commands to use keywords in different languages than english. Thus for instance, after entering the following rule the `Recommencer` command will correspond to `Restart`.

```
Coq < Grammar vernac vernac :=
Coq <                [ "Recommencer" "." ] -> [$0=<:vernac:<Restart.>>].
```

Example 3 : We can give names to repetitive tactic sequences. Thus in this example "`IntSp`" will correspond to the tactic `Intros` followed by `Split`.

```
Coq < Grammar tactic simple_tactic :=
Coq <                [ "IntSp" ] -> [$0=<:tactic:<Intros; Split>>].
```

Let us check that this works.

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

Coq < IntSp.
```

9.3.7 A word on grammar compiling

The choice of the sequence of grammar rules to use in the parsing of an expression is done according to an algorithm called the parsing method. This sequence should be unique otherwise we say that there is ambiguity. The parsing methods are classified according to the grammar class they accept as input. In our case, the method used is close to the $LL(1)$ method. The $LL(1)$ grammars are those for which we can choose the grammar rule to apply by seeing only the current token in the expression. There exists an algorithm to verify if a grammar is $LL(1)$ or not; it is based on the construction of two token sets *firsts* and *nexts* for each LMP.

In our case, we only construct the firsts set. The firsts set of a LMP is formed by the first tokens of the expressions it can recognize. It contains ϵ if the LMP can recognize the empty expression ϵ.

We are going to describe briefly the method used by Coq to verify the non-ambiguity of a grammar. If the grammar is non-ambiguous, it is transformed into a form called compiled grammar. This processes of verification and transformation is called *compilating*.

Compilating a grammar consists in factoring (i.e. taking the longest common factor) its LMP that have the same firsts sets.

We execute recursively this operation on the new LMP (i.e. the initial LMP without their common factor). There are two halting cases :

1. The LMP we process have the same firsts set but have no common factor. The grammar is refused.

2. All the LMP we process have different firsts sets. The grammar is accepted and its compiled form is the grammar with all the factorizations already performed. This grammar is stocked in the compiled grammar table.

When we extend a grammar with one or several rules, we should recompile it but also recompile all the grammars that mention it in their LMP. To avoid frequent recompilings, the new rules added are not immediately compiled but only stocked in the uncompiled grammars table. The grammars of this table are compiled when the system needs to consult the compiled grammars table, i.e. there is no recompilings during a parsing using primitive grammars. During a recompiling, the system prints the message `[Recompiling` n `nonterminal(s)...]` where n is the number of the grammars it recompiles.

Example : A trivial (and frequent) example of ambiguous grammar is a grammar with two identical LMP. The following rule has the same LMP that the function composition rule.

```
Coq < Grammar command command8 :=
Coq <                [ command7($A) "o" command8($B) ] ->
Coq <                [$0=<<($A /\ $B)>>].
```

This rule is not immediately compiled. It will be compiled when the system will do a parsing with non-primitive grammars. For instance, to parse the command "`Eval O *+ O.`", the system should recompile all the uncompiled grammars, `command command8` in our case. An error occurs and the rule is refused. The parsing does not fail and is done with the rules already compiled.

```
Coq < Eval O *+ O.
```

Let us comment on the error message. It indicates that the extended grammar `command command8` is not $LL(1)$ because after factorization, we obtain two empty `[]` LMP. These LMP have the same firsts set ($\{\epsilon\}$) but do not have a common factor. It is the first halting case : the extended grammar is refused.

More complicated cases of ambiguous grammars may arise. There is no universal solution : the user itself should transform its grammar to be accepted by the system. However, it does not have to remember all the rules entered in the system. Indeed the `Print Grammar` will do it for him.

```
Coq < Print Grammar command command8.
```

Note that the actions are printed as AST.

9.3.8 Limitations

The extensible grammar mechanism have two serious limitations.

- There is no command to remove a grammar rule. However there is a trick to do it. It is sufficient to execute the "`Reset`" command on a constant defined before the rule we want to remove. Thus we retrieve the state before the definition of the constant, then without the grammar rule.
- Grammar rules defined inside a section are automatically removed after the end of this section : they are available only inside it.

9.3.9 Extensible Grammar Syntax

It is possible to extend a grammar with several rules at once.

$$
\begin{array}{lll}
\texttt{Grammar}\ \mathit{univers}\ \mathit{nonterminal} & \texttt{:=} & \mathit{production}_1 \\
 & \texttt{|} & \vdots \\
 & \texttt{|} & \mathit{production}_n\ .
\end{array}
$$

Also, we can extend several grammar at the same time.

$$
\begin{array}{llllll}
\texttt{Grammar} & \mathit{univers} & \mathit{nonterminal}_1 & \texttt{:=} & \mathit{production}_1^1 \\
 & & & \texttt{|} & \vdots \\
 & & & \texttt{|} & \mathit{production}_{n_1}^1 \\
 & \texttt{with} & \vdots & & \\
 & \texttt{with} & \mathit{nonterminal}_p & \texttt{:=} & \mathit{production}_1^p \\
 & & & \texttt{|} & \vdots \\
 & & & \texttt{|} & \mathit{production}_{n_p}^p\ .
\end{array}
$$

We give the exact syntax for the extensible grammar mechanism. We use the BNF notation.

```
        ⟨Grammar⟩ ::= Grammar ⟨Entry⟩ ⟨Grammar_entry⟩ {with ⟨Grammar_entry⟩} .

          ⟨Entry⟩ ::= vernac | command | tactic | prim | ⟨Identifier⟩

  ⟨Grammar_entry⟩ ::= ⟨NonTerminal⟩ [⟨Parameters⟩] := [⟨Production⟩ {| ⟨Production⟩}]

    ⟨NonTerminal⟩ ::= ⟨Identifier⟩

     ⟨Parameters⟩ ::= [ [⟨Meta⟩ {;⟨Meta⟩}] ]

     ⟨Production⟩ ::= ⟨LMP⟩ -> ⟨Action⟩

            ⟨LMP⟩ ::= [ {⟨Production_item⟩} ]

⟨Production_item⟩ ::= " ⟨Token⟩ " | ⟨NonTerminalCall⟩

⟨NonTerminalCall⟩ ::= [⟨Entry⟩ :] ⟨NonTerminal⟩ [⟨Args⟩] [⟨Res⟩]

           ⟨Args⟩ ::= [ [⟨Action⟩ {; ⟨Action⟩}] ]

            ⟨Res⟩ ::= ( ⟨Meta⟩ )

         ⟨Action⟩ ::= [ [⟨Binding⟩ {; ⟨Binding⟩}] ]
                    |  case ⟨Meta⟩ of ⟨Case⟩ {| ⟨Case⟩} esac
                    |  ( ⟨Action⟩ )

        ⟨Binding⟩ ::= ⟨Meta⟩ = ⟨Pattern⟩

           ⟨Case⟩ ::= ⟨Pattern⟩ -> ⟨Action⟩
```

Figure 9.1: Extensible Grammar Syntax

Chapter 10

Writing your own pretty printing rules

10.1 Introduction

The V5.10 version of Coq provides a mechanism for extending the vernacular's parser and printer by adding, in an interactive way, new grammar and printing rules. The printing rules will be stocked into a table and will be recovered at the moment of the printing by the vernacular's printer.

The user can now define new constants, tactics and vernacular phrases with his desired syntax. The binding is dynamic. The printing rules for new constants should be written after the definition of the constant. This is to ensure that the symbols occurring in the pattern of the rule will be dynamically correctly bound. The rules should be outside a section if the user wants them to be exported.

The printing rules corresponding to the heart of the system (primitive tactics, commands and the vernacular language) are defined, respectively, in the files `PPTactic.v`, `PPCommand.v` and `PPVernac.v` (in the directory `src/syntax`). These files are automatically loaded by the file `main.ml` in the `src` directory. The user is not expected to modify these files unless he dislikes the way primitive things are printed, in which case he will have to compile the system after doing the modifications.

The system also uses the vernacular printer to report the vernacular phrases causing an error. When extending the printer, the error reporting mechanism is also implicitly extended. One way to test the printing rules for a certain phrase is to give it to Coq in a wrong environment, just to look at the reported error message. When the system fails to find a suitable printing rule, a tag `#GENTERM` appears in the message.

In the following we give some examples showing how to write the printing rules for the non-terminal and terminal symbols of a grammar . We will test them frequently by inspecting the error messages. Then, we give the grammar of printing rules and a description of its semantics. The syntax of the patterns that can appear in either grammar or printing rules is described in section 10.4.

10.2 The Printing Rules

10.2.1 The printing of non terminals

The printing is the inverse process of parsing. While a grammar rule maps an input string into an abstract syntax tree (ast), a printing rule maps an ast into an output string. So given a certain grammar rule, the printing rule can be obtained by inverting the grammar rule.

A printing rule is of the form :

`Syntax` *universe name DPattern precedence printing_order rec_bindings*.

where :

- *universe* is an identifier denoting the universe of the ast to be printed. There is a correspondence between the universe of the grammar rule used to generate the ast and the one of the printing rule :

Univ. Grammar	*Univ. Printing*
vernac	vernac
tactic	tactic
command	constr

- *name* is an identifier corresponding to the name of the printing rule. A rule is identified by both its universe and name, if there are two rules with both the same name and universe, then the last one overrides the former.

- *DPattern* is a pattern that matches the ast to be printed. The syntax of patterns is very similar to the patterns for grammar rules. A description of their syntax is given in section 10.4.

- *precedence* is positive integer indicating the precedence of the rule. In general the precedence for tactics and vernacular phrases is 0. The universe of commands is implicitly stratified by the hierarchy of the parsing rules. We have non terminals *command*0 , *command*1, etc. The idea is that objects parsed with the non terminal $command_i$ have precedence i. In most of the cases we fix the precedence of the printing rules for commands to be the same number of the non terminal with which is parsed.

- *printing_order* is the sequence of orders indicating the concrete layout of the printer.

- *rec_bindings* is used to deal with recursion in the printing rules and it is optional.

Example 1 : *Defining the syntax for new tactics*
Let's see the production of a new tactic `MyExact` with the same syntax as the primitive tactic `Exact` :

```
Coq < Grammar tactic simple_tactic :=
Coq <   [ "MyExact" comarg($c) ] -> [$0 = (MyExact $c)].
```

If we try to use `MyExact 0` the system reports an error with the tag `#GENTERM` appearing in it :

```
Coq < MyExact O.
```

The vernacular's printer does not know how to print that phrase. Considering that printing rules for objects of `comarg` have already been defined, let's see a possible rule for our tactic `MyExact` :

```
Coq <
Syntax tactic myexact <:tactic: <MyExact $c>> 0 "MyExact "<$c:"CommandArg":*>.
```

The universe of the tactics is `tactic` and the name of the rule is `myexact`. Between `<:tactic: <` and `>>` we are allowed to use the syntax of tactics. The system will call the parser of tactics to determine the structure of the ast. `0` is the precedence for tactics.

The printing order `"MyExact " <$c:"CommandArg":*>` tells to print the string `MyExact` followed by its command argument. The string `"CommandArg"` gives information about the value of `$c` and it is just for documentation purposes. The * tells not to put parentheses around the value of `$c`.

Now if we try `MyExact O`. We see it is well printed in the error message.

```
Coq < MyExact O.
```

Another way to obtain the printing rule is by inverting the grammar production using exactly the same pattern of the grammar rule :

```
Coq < Syntax tactic myexact (MyExact $c) 0 "MyExact "<$c:"CommandArg":*>.
```

Example 2 : *Defining the syntax for new constants.*
Let's define the constant `Xor` in Coq :

```
Coq < Definition Xor := [A,B:Prop] A/\~B \/ ~A/\B.
```

Given this definition, we may want to use the syntax of `A X B` to denote `(Xor A B)`. To do that we give the grammar rule :

```
Coq < Grammar command  command7 :=
Coq <    [ command6($c1) "X" command7($c2) ] -> [$0 = <<(Xor $c1 $c2)>>].
```

Note that the operator is associative to the right. Now `True X False` is well parsed :

```
Coq < Goal True X False.
```

To have it well printed we extend the printer :

```
Coq < Syntax constr Pxor  <<(Xor  $t1  $t2)>>  7
Coq <       <$t1:"term":L>  " X "   <$t2:"term":E>.
```

and now we have the desired syntax :

```
Coq < Show.
```

Let's comment the rule :

- constr is the universe of the printing rule.
- Pxor is the name of the printing rule.
- <<(Xor $t1 $t2)>> is the pattern of the ast to be printed. Between << >> we are allowed to use the syntax of command. Metavariables may occur in the pattern but preceded by $.
- 7 is the rule's precedence and it is the same one than the parsing production (command7).
- <$t1:"term":L> " X " <$t2:"term":E> are the printing orders, it tells to print the value of $t1 then the symbol X and then the value of $t2.

  The L in the little box <$t1:"term":L> indicates not to put parentheses around the value of $t1 if its precedence is **less** than the rule's one. An E instead of the L would mean not to put parentheses around the value of $t1 if its the precedence is **less or equal** than the rule's one. In the example before we saw that with the option * no parenthesis are written around the value of $t1.

  The associativity of the operator can be expressed in the following way :

  <$t1:"term":L> " X " <$t2:"term":E> associates the operator to the right.

  <$t1:"term":E> " X " <$t2:"term":L> associates to the left.

Note that while grammar rules are related by the name of non-terminals (command6 and command7) printing rules are isolated. The *Pxor rule* tells how to print an "Xor expression" but not how to print its subterms. The printer looks up recursively the rules for the values of $t1 and $t2. The selection of the printing rules is strictly determined by the structure of the ast to be printed.

Example 3 : *Forcing to parenthesize a new syntactic construction*
You can force to parenthesize a new syntactic construction by fixing the precedence of its printing rule to a number greater than 9. For example a possible printing rule for the Xor connector in the prefix notation would be :

```
Coq < Syntax constr ex_imp <<(Xor $t1  $t2)>> 10
Coq <             "X "  <$t1:"term":L> " "   <$t2:"term":L> .
```

No explicit parentheses are contained in the rule, nevertheless, when using the connector, the parentheses are automatically written :

```
Coq < Show.
```

A precedence higher than 9 ensures that the ast value will be parenthesized by default in either the empty context or if it occurs in a context where the instructions are of the form <$t:"string":L> or <$t:"string":E>.

Example 4 : *Dealing with list patterns in the syntax rules*
The following productions extend the parser to recognize a tactic called MyIntros that receives a list of identifiers as argument as the primitive Intros tactic does :

```
Coq < Grammar tactic my_ne_identarg_list :=
Coq <   [ identarg($id) my_ne_identarg_list($idl) ] -> [$0 = ($CONS $id $idl)]
Coq < | [ identarg($id) ] -> [$0 = ($LIST $id)].


Coq < Grammar tactic simple_tactic :=
Coq <   [ "MyIntros" my_ne_identarg_list($idl) ] ->
Coq <                   [$0 =($OPER{MyIntrosWith} $idl)].
```

The non terminal `my_ne_identarg_list` defines the non-empty lists of identifiers. The patterns `($CONS $id $idl)` and `($LIST $id)` are list patterns. The former denotes a list pattern of *at least one element*, and the latter a list of *exactly one element*. The list pattern `($LIST)` and `($NIL)` denote the *empty list*. Note that both the patterns `($CONS $id $idl)` and `($LIST $id)` may denote a list of only one element.

To define the printing rule for `MyIntros` it is necessary to define the printing rule for the non terminal `my_ne_identarg_list`. In grammar productions the dependency between the non terminals is explicit. This is not the case for printing rules, where the dependency between the rules is determined by the structure of the pattern. So, the way to make explicit the relation between printing rules is by adding structure to the patterns.

```
Coq < Syntax tactic myintroswith ($OPER{MyIntrosWith} $L) 0
Coq <     "MyIntros " <$IDLIST:"identifiers":*>
Coq <     with $IDLIST:=($OPER{MYNEIDENTARGLIST} $L).
```

This rule says to print the string `MyIntros` and then to print the value of `$IDLIST`. This variable is bound to the pattern `($OPER{MYNEIDENTARGLIST} $L)`. This is an example of printing rule with bindings, in this case there is only one but there may be an arbitrary list of bindings after the `with`.

The operator `$OPER{<id>}` injects a list pattern into patterns. The name of the injection `MYNEIDENTARGLIST`, was arbitrarily selected. The following rules indicate how to print an ast with that structure :

```
Coq < Syntax tactic my_ne_identarg_list_cons
Coq <      ($OPER{MYNEIDENTARGLIST} ($CONS $id $l)) 0
Coq <           <$id :"Ident":*> " " <$TAIL:"Tail":*>
Coq <            with $TAIL := ($OPER{MYNEIDENTARGLIST} $l).
Coq <
Coq < Syntax tactic my_ne_identarg_list_single
Coq <   ($OPER{MYNEIDENTARGLIST} ($LIST $id))  0
Coq <         <$id :"Ident":*>.
```

The first rule says how to print a non-empty list, while the second one says how to print the list with exactly one element. Note that the pattern structure of the binding in the first rule ensures its use in a recursive way.

While the order of grammar productions is not relevant, the order of printing rules is. In case of two rules whose patterns superpose each other the **last** rule is always chosen. In the example, if the last two rules were written in the inverse order the printing will not work, for only the rule *my_ne_identarg_list_cons* would be recursively retrieved and there is no rule for the empty list. Other possibilities would have been to write a rule for the empty list instead of the *my_ne_identarg_list_single* rule.

```
Coq < Syntax tactic my_ne_identarg_list_nil ($OPER{MYNEIDENTARGLIST} ($LIST))  0.
```

This rule indicates to do nothing in case of the empty list. In this case there is no superposition between patterns (no critical pairs) and the order is not relevant.

Example 5 : *Defining constants with arbitrary number of arguments*

Sometimes the constants we define may have an arbitrary number of arguments, the typical case are polymorphic functions. Let's consider for example the composition operator presented in the documentation of grammars defined by :

```
Coq < Definition explicit_comp := [A,B,C:Set][f:A->B][g:B->C] [a:A] (g (f a)).
```

The following rule extend the parser :

```
Coq < Grammar command command6 :=
Coq <
[command5($c1)  "o" command6($c2) ] -> [$0=<<(explicit_comp ? ? ? $c1 $c2)>>].
```

Our first idea is to write the printing rule just by "inverting" the production :

```
Coq < Syntax constr pp_comp <<(explicit_comp ? ? ? $f $g)>> 6
Coq <         <$f:"Term":L> "o" <$g:"Term":L>.
```

This rule is not correct : ? is not allowed as a metavariable identifier for patterns in printing rules. If we had used the pattern `<<(explicit_comp $_ $_ $_ $f $g)>>` instead, the rule will be used only in rare cases : when the values associated to each occurrence of `$_` are the same. The reason is that `$_` does not denote an anonymous metavariable.

The process of matching an ast with a pattern tests that all the ast values associated to the same metavariable in the pattern are the same. There is **no syntax** for denoting anonymous metavariables in patterns of printing rules. This means that, for every metavariable occurring several times in the pattern, this test is done. In particular, for the identifier `$_`. This is an important difference between the syntax of patterns in grammar rules and in printing rules. Here is a correct version of this rule :

```
Coq < Syntax constr pp_comp <<(explicit_comp $1 $2 $3 $f $g)>> 6
Coq <         <$f:"Term":L> "o" <$g:"Term":L>.
```

Let's test the printing rule :

```
Coq < Definition Id := [A:Set][x:A]x.

Coq < Eval (Id nat) o (Id nat).

Coq < Eval ((Id nat)o(Id nat)  O).
```

In the first case the rule was used, while in the second one the system failed to match the pattern of the rule with the ast of `((Id nat)o(Id nat) O)`. Internally the ast of this term is the same as the ast of the application `(explicit_comp nat nat nat (Id nat) (Id nat) O)` . When the system retrieves our rule it tries to match an application of six arguments with an application of five arguments (the ast of `(explicit_comp $1 $2 $3 $f $g)`). Then, the matching fails and the term is printed using the rule for application.

Note that the idea of adding a new rule for `explicit_comp` for the case of six arguments does not solve the problem, because of the polymorphism, we can always build a term with one argument more. The rules for application deal with the problem of having an arbitrary number of arguments by using list patterns. Let's see these rules :

```
Coq < Syntax constr   app ($OPER{APPLIST} ($CONS $H $T)) 10
Coq <     [<hov 0>  <$H:"Function":E>  <$P2:"Argument":E>  ]
Coq <     with $P2:=($OPER{APPTAIL} $T).
Coq <
Coq < Syntax constr   apptailcons ($OPER{APPTAIL} ($CONS $H $T)) 10
Coq <     [1 1] <$H:"Arg":L>  <$TL:"Tail":E> with $TL:=($OPER{APPTAIL} $T).
Coq < Syntax constr   apptailnil ($OPER{APPTAIL} ($LIST)) 10.
```

The first rule prints the operator of the application, and the second prints the list of arguments. Then, one solution to our problem is to specialize the first rule of the application to the cases where the operator is `explicit_comp` and the list pattern has **at least** five arguments :

```
Coq < Syntax constr pp_comp
Coq <     ($OPER{APPLIST} ($CONS (CONST {#explicit_comp.cci}) ($CONS $1 ($CONS $2
Coq <                          ($CONS $3 ($CONS $f  ($CONS $g $l)))))))) 10
Coq <          <$f:"Term":L>  "o" <$g:"Term":L> " " <$L:" ":*>
Coq <          with $L := ($OPER{APPTAIL} $l).
```

Now we can see that this rule works for any application of the operator :

```
Coq < Eval ((Id nat) o (Id nat) O).
Coq < Eval ((Id nat->nat) o (Id nat->nat) [x:nat]x O).
```

In the examples presented by now, the rules have no information about how to deal with indentation, break points and spaces, the printer will write everything in the same line without spaces. To indicate the concrete layout of the patterns, there's a simple language of printing instructions that will be described in the following section.

10.2.2 The printing of terminals

The user is not expected to write the printing rules for terminals, this is done automatically. Primitive printing is done for :

- arguments of the `$PRIM` operator. The grammar `prim` yields ast values that can be decomposed by patterns of the form (`$PRIM` $\$id$) , then the printing of the value associated to $\$id$ is done automatically.

  Let's see for example the rules for `MyCd` :

```
Coq < Grammar vernac vernac :=
Coq <   [ "MyCd" prim:string($dir) "." ] -> [$0 = (MYCD $dir)].
```

If we write the naive rule :

```
Coq < Syntax vernac mycd (MYCD $dir) 0
Coq <      "MyCd "  <$dir:"string":*>.
```

It will not work :

```
Coq < MyCd "dir".
```

The metavariable `$dir` is bound to an ast value that should still be destructured by a pattern having a `$PRIM` :

```
Coq < Syntax vernac mycd (MYCD ($PRIM $dir)) 0
Coq <      "MyCd "  <$dir:"string":*>.
```

Now the result is correct :

```
Coq < MyCd "dir".
```

Sometimes printing rules may be different depending whether the terminal has been parsed by `prim:string` or `prim:ident`, etc. For that there is a way to destructure a terminal with `$PRIM` specifying the desired injection (or "type"). The possible injections are `INT`, `STRING`, `PATH`, `IDENT` or `DYN`.

In the example of `MyCd` we would have written the rule with the injection `STRING`.

```
Coq < Syntax vernac mycd (MYCD ($PRIM $dir (SOME {STRING}))) 0
Coq <      "MyCd "  <$dir:"string":*>.
```

- The ast values with pattern structure (`$VAR $id`).

  For example, given the grammar rule :

```
Coq < Grammar tactic identarg := [ prim:ident($id) ] -> [$0 = ($VAR $id)].


Coq < Grammar tactic simple_tactic :=
Coq <   [ "MyIntro" identarg($id)] -> [$0 = (MyIntrosWith $id)].
```

The following printing rule is correct :

```
Coq < Syntax tactic myintroswith (MyIntrosWith $id) 0
Coq <         "Intro " <$id:"identifier":*>.
```

The system knows how to print an ast value having the structure (`$VAR` *value*).

10.3 Syntax for pretty printing rules

In the following we give the syntax for printing rules. The metalanguage conventions are the same as those specified for the definition of the *Pattern*'s syntax in section 10.4.

Printing − Rule ::=
    `Syntax` *ident* *ident* *DPattern* *precedence* *printing_order** *rec_bindings* .

are: *precedence* ::= *int* | `[` *int* *int* *int* `]`

rec_bindings ::= ϵ | `with` *binding*$^+$

binding ::= *metav* `:=` *patt*

printing_order ::=
  `FNL`
  | *string*
  | `[` *int int* `]`
  | `[` *box printing_order** `]`
  | `<` *metav* `:` *string* `:` *paren_rel* `>`

box ::= `<` *box_type int* `>`

box_type ::= `hov` | `hv` | `v` | `h`

paren_rel ::= `*` | `L` | `R`

DPattern is **almost** the same set of patterns defined by *Pattern**. The main differences are :

(a) there is no syntax for anonymous metavariables (`$_` is just a common identifier).

(b) there is a new kind of pattern that allows to destructure ast the values generated by the grammars `prim`. These patterns are of the form :
  `($PRIM` *metav* `)`
  `($PRIM` *metav* `(SOME {` *inj* `}))`
  where *inj* may be `INT`, `STRING`, `PATH`, `IDENT`, `DYN`.

(c) the operator `$APPEND` is not available any more.

Note that while patterns in printing rules are destructive, patterns in the bindings of the printing rules are constructive.

10.3.1 Pretty grammar structures

The basic structure is the printing order sequence. Each order has a printing effect and they are sequentially executed. The orders can be :

*see the description of *Pattern* in section 10.4

– printing orders
– printing boxes

Printing orders

Printing orders can be of the form :

- "*string* " prints the *string*.
- `FNL` force a new line.
- < \$*id* : *comment* : *paren_rel* > at the moment of the printing, \$*id* is bound to an ast value. The printer looks up the adequate printing rule for that ast value and applies recursively this method. Recursion of the printing is determined by the pattern's structure. *comment* is just an arbitrary string used for documentation purposes. If *t* is the ast value associated to \$*id*, then the meaning of *paren_rel* is the following :

  `L` if *t* 's precedence is **less** than the rule's one, then no parentheses around *t* are written.
  `E` if *t* 's precedence is **less or equal** than the rule's one then no parentheses around *t* are written.
  `*` **never** write parentheses around *t*.

Printing boxes

The concept of formatting boxes is used to describe the concrete layout of patterns : a box may contain many objects which are orders or subboxes sequences separated by breakpoints; the box wraps around them an imaginary rectangle.

1. **Box types**

   The type of boxes specifies the way the components of the box will be displayed and may be :

   – `h` : to catenate objects horizontally.
   – `v` : to catenate objects vertically.
   – `hv` : to catenate objects as with an "h box" but an automatic vertical folding is applied when the horizontal composition does not fit into the width of the associated output device.
   – `hov` : to catenate objects horizontally but if the horizontal composition does not fit, a vertical composition will be applied, trying to catenate horizontally as many objects as possible.

   The type of the box can be followed by a *n* offset value, which is the offset added to the current indentation when breaking lines inside the box.

2. **Boxes syntax**

   A box is described by a sequence surrounded by `[ ]`. The first element of the sequence is the box type : this type surrounded by the symbols `< >` is one of the words `hov`, `hv`, `v`, `v` followed by an offset. The default offset is 0 and the default box type is `h`.

3. **Breakpoints**

   In order to specify where the pretty-printer is allowed to break, one of the following break-points may be used :

   - `[0 0]` is a simple break-point, if the line is not broken here, no space is included ("Cut").
   - `[1 0]` if the line is not broken then a space is printed ("Spc").
   - `[i j]` if the line is broken, the value j is added to the current indentation for the following line; otherwise blank spaces are inserted ("Brk").

   Examples : It is interesting to test printing rules on "small" and "large" expressions in order to see how the break of lines and indentation are managed. Let's define two constants and make a `Print` of them to test the rules. Here are some examples of rules for our constant `Xor` :

```
Coq < Definition A := True X True.

Coq < Definition B := True X True X True X True X True X True X True
Coq <                  X True X True X True X True X True X True.


Coq < Syntax constr Pxor  <<(Xor  $t1  $t2)>>  6
Coq <         <$t1:"term":L>  " X "   <$t2:"term":E>.
```

   This rule prints everything in the same line exceeding the line's width.

```
Coq < Print B.
```

   Let's add some break-points in order to force the printer to break the line before the operator :

```
Coq < Syntax constr Pxor  <<(Xor  $t1  $t2)>>  6
Coq <         <$t1:"term":L> [0 1] " X "    <$t2:"term":E>.


Coq < Print B.
```

   The line was correctly broken but there is no indentation at all. To deal with indentation we use a printing box :

```
Coq < Syntax constr Pxor  <<(Xor  $t1  $t2)>>  6
Coq <        [<hov 0>  <$t1:"term":L> [0 1] " X "   <$t2:"term":E> ].
```

   With this rule the printing of A is correct, an the printing of B is indented.

```
Coq < Print B.
```

   If we had chosen the mode `v` instead of `hov` :

```
Coq < Syntax constr Pxor  <<(Xor  $t1  $t2)>>  6
Coq <        [<v 0>  <$t1:"term":L> [0 1] " X "   <$t2:"term":E> ].
```

We would have obtained a vertical presentation :

```
Coq < Print A.
```

The difference between the presentation obtained with the `hv` and `hov` type box is not evident at first glance. Just for clarification purposes let's compare the result of this silly rule using an `hv` and a `hov` box type :

```
Coq < Syntax  constr Pxor  <<(Xor  $t1  $t2)>>  6
Coq <     [<hv 0>  "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
Coq <       [0 0]   "-----------------------------------------"
Coq <       [0 0]   "ZZZZZZZZZZZZZZZZ" ].


Coq < Print A.


Coq < Syntax  constr Pxor  <<(Xor  $t1  $t2)>>  6
Coq <     [<hov 0>  "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
Coq <       [0 0]   "-----------------------------------------"
Coq <       [0 0]   "ZZZZZZZZZZZZZZZZ" ].


Coq < Print A.
```

In the first case, as the three strings to be printed do not fit in the line's width, a vertical presentation is applied. In the second case, a vertical presentation is applied, but as the last two strings fit in the line's width, they are printed in the same line.

10.4 Pattern's syntax

The grammar rules maps an input string into an abstract syntax tree (ast), while the printing, conversely, maps an output string into an ast. To describe this mapping, both grammar and printing rules need some syntax to denote an ast. That concrete syntax is what we call *Pattern*.

The patterns are conceptually divided into two classes : constructive patterns (those that can be used in parsing rules) and destructive patterns (those that can be used in pretty printing rules). In the following we give the concrete syntax of patterns and some examples of their usage.

The grammar † *Pattern* defines the syntax of both constructive and destructive patterns.

†The metasymbols we will use have the following meaning :
x^* : 0 or more occurrences of x.
x^+ : 1 or more occurrences of x.
$s_1 - s_n$: any of the symbols in the range from s_1 to s_n.

Pattern ::= << *metav_command* >>
| <:command: < *metav_command* >>
| <:vernac: < *metav_vernac* >>
| <:tactic: < *metav_tactic* >>
| *patt*

patt ::= *ident*
| *token*
| { *token* }
| (*ident* *patt**)
| [<>] *patt*
| [*ident*] *patt*

token ::= *int* | *ident* | *string* | *path*

path ::= (# *ident*)$^+$. *univ*

metav_command, *metav_vernac*, *metav_tactic*, stand, respectively, for the syntax of commands, vernacular phrases and tactics. The prefix *metav* is just to emphasize that identifiers beginning with $ denote metavariables.

int is a sequence of digits, *string* is any sequence of characters delimited by " " . The set *ident* can be defined by the regular expression ‡ :

($ | _ | a − z | A − Z) ($ | _ | a − z | A − Z | 0 − 9)*

So, a pattern can be either an identifier, a token, an application or an abstraction (either binding or non-binding). Identifiers begining with the symbol $ denote metavariables, their value will be calculated when using the pattern either in the parsing, or in the printing. There are some identifiers that have a special meaning and should be used in a certain way. The system makes no control at the moment of the parsing to test that those identifiers are correctly used. In general, errors are detected at the moment of using the patterns. The following list of special patterns aims to give an idea of their common usage :

‡Identifiers can be also any sequence of characters delimited by simple quotes ' '.

list_pattern_pat
($VAR *metav*)
($PRIM *metav*)
($PRIM *metav* (SOME { *inj* })
($OPER{ *ident* } *list_pattern_patt*)
($QUOTE *patt*)
($SLAML *list_pattern_someid* *patt*)

list_pattern_patt ::=
metav
| ($NIL)
| ($LIST *patt**)
| ($CONS *patt* *list_pattern_patt*)
| ($APPEND *list_pattern_patt**)

list_pattern_someid ::=
metav
| ($NIL)
| ($LIST *someid**)
| ($CONS *someid* *list_pattern_someid*)
| ($APPEND *list_pattern_someid**) *someid* ::= *metav* | (SOME *id*) | NONE

inj ::= INT | STRING | PATH | IDENT | DYN

The operator `$APPEND` is only for constructive patterns while `$PRIM` is for destructive ones.

Let's show the use of some of these patterns, more examples can be found in the sections describing the grammar and printing.

- `$VAR` is an injection applied to identifiers parsed by the `prim:ident` grammar. Generally it is used to inject identifiers into commands :

```
Coq < Grammar tactic identarg :=
Coq <    [ prim:ident($id) ] -> [$0 = ($VAR $id)].
```

- The elements of *metav* are identifiers begining by `$`, they denote metavariable and will be bound to an ast value at the moment of the parsing or the printing.

- The operator `$CONS` builds a list pattern from a pattern and a list pattern. The operator `$LIST` builds a list pattern from a possible empty sequence of patterns. The pattern `($LIST)` denotes the empty list as well as `($NIL)`. `$APPEND` takes a possible empty sequence of list patterns and returns a list pattern.

  There may be several patterns to denote an ast. For example, to denote a list pattern of exactly n elements, we can write :

```
($CONS $p1 ($CONS $p2 (...($CONS $pn ($NIL))..))
($CONS $p1 ($CONS $p2 (...($CONS $pn ($LIST))..))
($CONS $p1 ($CONS $p2 (...($CONS $pn-1 ($LIST $pn))..))
($LIST $p1 $p2 ... $pn)
($APPEND ($LIST $p1) ($LIST $p2) ($LIST $p3...$pn))
```

The production corresponding to a non empty list of indentifiers uses this kind of patterns :

```
Coq < Grammar tactic ne_identarg_list :=
Coq <   [ identarg($id) ne_identarg_list($idl) ] -> [$0 = ($CONS $id $idl)]
Coq < | [ identarg($id) ] -> [$0 = ($LIST $id)].
```

- The operator `$OPER` allows to inject a list pattern into ast patterns. In the expression `($OPER{` *id* `} (` *list_pattern_patt* `))` the identifier *id* is the name of the injection and tags the list pattern.

  The tactic `Intros` takes a list of identifiers as argument, and its parsing rule uses `$OPER` :

```
Coq < Grammar tactic simple_tactic :=
Coq <  [ "Intros" ne_identarg_list($idl) ] -> [$0 =($OPER{IntrosWith}$idl)].
```

- `[<>]`*patt* and `[`*metav*`]`*patt* are patterns for abstractions. The former denotes a non-binding abstraction, and the latter a binding one. The productions corresponding to the non-dependent product and to the lambda abstraction use these kind of patterns :

```
Coq < Grammar command command8 :=
Coq <   [ command7($c1) "->" command8($c2) ] -> [$0 = (PROD $c1 [<>]$c2)].


Coq < Grammar command command0 :=
Coq <    [ "[" ident($id1) ":" command($c) "]" command($c2) ]
Coq <   -> [$0 = (LAMBDA $c [$id1]$c2)].
```

- `($SLAML l body)` is used to denote an abstraction where the elements of the list pattern `l` are the variables simultaneously abstracted in `body`.

  The production to recognize a lambda abstraction of the form $[x_1, \ldots, x_n : T]body$ use the operator `$SLAM` :

```
Coq < Grammar command command0 :=
Coq <   [ "[" ident($id1) "," binder($idl) ":" command($c) "]" command($c2) ]
Coq <      -> case $idl of
Coq <             ($OPER{IDBINDER} $L) ->
Coq <                [$0 = (LAMBDALIST $c ($SLAML ($CONS (SOME $id1) $L) $c2))]
Coq <           esac.
```

10.5 Debugging the printing rules

By now, there is almost no semantic check of printing rules in the system. To find out where the problem is, there are two possibilities : to analyze the rules looking for the most common errors or to work in the toplevel tracing the ml code of the printer.

10.5.1 Most common errors

Here are some considerations that may help to get rid of simple errors :

- make sure that the rule you want to use is not defined in previously closed section.
- make sure that **all** nonterminals of your grammar have their corresponding printing rules.
- make sure that *there is no free occurrence* of a metavariable in a rule. For example if you enter this rule in Coq :

```
Coq < Syntax constr Pxor  <<(Xor  $t1  $t2)>>  6
Coq <         <$T1:"term":E>  " X "    <$t2:"term":L>.
```

  `$T1` is free but the system accepts this rule without giving any warning. At the moment of using it, the system raises a message :

```
Coq < Print A.
```

- make sure that the set of printing rules for a certain non terminal covers all the space of ast values for that non terminal.
- the order of the rules is important. If there are two rules whose patterns superpose (they have common instances) then it is always the last rule that will be retrieved.
- if there are two rules with the same name and universe the last one overrides the first one. The system always warns you about redefinition of rules.

10.5.2 Tracing the ml code of the printer

Some of the conditions presented above are not easy to verify when dealing with many rules. In that case tracing the ml code helps to understand what is happening. The printers are in the file `printer.ml` in the `src` directory. There you will find the functions :

- *genvernacpr*: the printer of the vernacular language
- *gencompr* : the printer of commands
- *gentacpr* : the printer of tactics

These printers are defined in terms of a general printer *genprint* by instantiating it with the adequate parameters.

genprint waits for : the precedence of the ast to print, the universe to which this ast belongs (*tactic*, *constr*, *vernac*), a printer for the tokens, a default printer and the ast to print. *genprint* looks up, in the table of rules, the rules that are necessary to print the ast and its subterms.

An ast of a universe may have subterms that belong to another universe. For instance, let v be the ast of the vernacular expression `MyExact O`. The function *genvernacpr* is called to print v. This function instantiates the general printer *genprint* with the universe *vernac*. Note that v has a subterm c corresponding to the ast of `O` (c belongs to the universe *constr*). *genprint* will try recursively to print all subterms of v as belonging to the same universe of v. If this is not possible, because the subterm belongs to another universe, then the default printer that was given as argument to *genprint* is applied. The default printer is responsible for changing the universe in a proper way calling the suitable printer for c.

Technical Remark. In the file `PPVernac.v` and `PPTactic.v`, there are some rules that do not arise from the inversion of a parsing rule. They are strongly related to the way the printing is implemented.

```
Coq < Syntax vernac constr (COMMAND $c) 8
Coq <         <$C:"Command":E> with $C:=(PPUNI$COMMAND $c).

Coq < Syntax vernac tactic (TACTIC $t) 100
Coq <         <$T:"Tactic":E> with $T:=(PPUNI$TACTIC $t).
```

As an ast of vernac may have subterms that are commands or tactics these rules allow the printer of vernac to change the universe. The `PPUNI$COMMAND` and `PPUNI$TACTIC` are special identifiers used for this purpose. They are used in the code of the default printer that *genvernacpr* gives to *genprint*.

The following rule is the analogue rule one for the universe of tactics.

```
Coq < Syntax tactic command (COMMAND $c) 8
Coq <       <$C:"Command":E> with $C:=(PPUNI$COMMAND $c).
```

Chapter 11

Writing tactics in Coq

Introduction

This chapter concerns advanced users who want to write an implementation in the `Coq` system. We do not intend to present the internal machinery of the whole system but we want to give here the basic notions of tactic writing. We will illustrate these notions with a very simple tactic — called `Mytactic` — which instantiates a universal hypothesis.

Our aim is to show that tactic writing is a "high level job" which does not presuppose a knowledge of the whole system, and certainly not a hard task left to some "wizards". Consequently, we will not detail the structure of the different types, which will be tedious, but we will just give their location and their meaning. We will notice that abstraction generally allows us to ignore these definitions.

Situation. We will suppose the reader to be familiar with the use of `Coq` and Caml Light. In the following, let `COQTOP` be the directory in which `Coq` is installed. Files names will be given relatively to this directory.

The main directories in `COQTOP` are the following :

`src`	`Coq` sources, shared among subdirectories `meta`, `constr`, `proofs`, `env`, `tactics` and `link`.
`src/lib`	Some Caml Light utilities.
`src/syntax`	The initial syntax of Coq.
`tactics`	Sources and vernacular entries of some tactics, like `Tauto`, `ProPre` or `Program`.
`theories`	The `Coq` files. The basic files of the system are in `theories/INIT`.
`MYTACTIC`	The directory in which we are going to write our tactic.

A very simple example. Let us start with a very simple tactic. Suppose we want to create a new tactic that is an abbreviation for the command `Contradiction Orelse Auto.` We can do it only with syntactic commands :

```
Coq < Grammar tactic simple_tactic :=
Coq <          [ "Autoplus" ] -> [$0 = <:tactic:<Contradiction Orelse Auto >>].
```

```
Coq < Syntax tactic Autoplus_rule <:tactic:<Autoplus>> 0 "Autoplus".
```

See sections related to `Grammar` and `Syntax` to understand the syntax of these commands. Just notice that we used the non-terminal `simple_tactic` (and not the `vernac` one for example) so `Autoplus` is a tactic and not a vernac command. As a consequence, `Autoplus` can be used inside tacticals like `Try`, `Orelse`,... Let us use our new tactic on an example :

```
Coq < Lemma foo : (A:Prop)False->A.
```

```
Coq < Autoplus.
```

```
Coq < Save.
```

Notice also that without the `Syntax` command, the tactic `Autoplus` would be printed as `Contradiction Orelse Auto` instead of `Autoplus` (during the printing of proof scripts or error messages).

Of course, from the moment we want to write more complex tactics (dealing with the structure of terms, looking in the environment, performing reductions,...) we need to write them in Caml Light and to add them into the system. In the following we explain all the steps of such a development.

Section 11.1 describes the representation of terms and basic operations on these terms (substitution, application, reduction, ...). Section 11.2 introduces the notion of tactic and gives tools to handle terms inside a tactic. In section 11.3 we show how to register a tactic (addition in the table of tactics, grammar's entry and pretty printing). Then we give a complete example in the section 11.4 (Caml Light code, registration and use). The last section describes some tools for debugging tactics.

11.1 Terms

11.1.1 Representation

The type `constr` of the terms of Calculus of Constructions is defined in `src/meta/generic.mli` and `src/constr/term.mli`.

First, a generic type `term` for terms is defined in `src/meta/generic.mli`, abstracted over the type `oper` of operators. There are four main constructors for terms :

- `VAR` *id*, a reference to a global variable of name *id*;
- `Rel` *n*, a variable in the de Bruijn notation;
- `DLAM` *t*, a de Bruijn binder on the term *t*;
- `DOPN (`*op args*`)`, the application of the operator *op* to the arguments *args*.

For reasons of efficiency, some of these constructors are duplicated :

- `DOP0` for operators of arity 0, `DOP1` for those of arity 1, `DOP2` for those of arity 2. `DOPL` is used to give arguments as a list instead of a vector (since all uses of `DOPL` fall into one of the previous categories, `DOPL` is not used in the core system. It is left for those who wish to experiment with the system).
- `DLAMV` for de Bruijn binder on many terms.

It leads to the following type :

```
type 'oper term =
    DOP0 of 'oper                              (* atomic terms *)
  | DOP1 of 'oper * 'oper term                 (* operator of arity 1 *)
  | DOP2 of 'oper * 'oper term * 'oper term    (* operator of arity 2 *)
  | DOPN of 'oper * 'oper term vect            (* operator + arguments' vector *)
  | DOPL of 'oper * 'oper term list            (* operator + arguments' list *)
  | DLAM of name * 'oper term                  (* de Bruijn binder on one term*)
  | DLAMV of name * 'oper term vect            (* de Bruijn binder on many terms*)
  | VAR of identifier                          (* named variable *)
  | Rel of int                                 (* variable as de Bruijn index *)
;;
```

In the binders the `name` is either `Name` *id*, where *id* is an identifier, or `Anonymous`, and is just kept for pretty printing. The type of identifiers is an abstract type `identifier` (see `src/meta/names.ml`). The functions

```
value id_of_string : string -> identifier;;
value string_of_id : identifier -> string;;
```

realize the conversion between the types `identifier` and `string`.
Then, the type `oper` of the operators of the Calculus of Constructions is defined in `src/constr/-term.mli`. The main constructors are :

```
type 'a oper =
    Sort of 'a              (* sorts        (DOP0) *)
  | Prod                    (* product      (DOP2) *)
  | Lambda                  (* abstraction  (DOP2) *)
  | AppL                    (* application  (DOPN) *)
  | Const of section_path   (* constants    (DOPN) *)
  | Cast                    (* cast         (DOP2) *)
  | ...
```

`'a` is the type of sorts. The sorts are here {Prop, Set, Type, Typeset}. Prop and Type are sorts for logical propositions, and Set and Typeset for propositions with an informative content (specifications). The sorts Type and Typeset contain an universes hierarchy implicitly managed by the system. The corresponding type is :

```
type contents = Pos | Null;;
type sorts =
    Prop of contents
  | Type of contents * impuniv__universe;;
```

with the four possibilities :

`Prop Null`	Prop
`Prop Pos`	Set
`Type Null,_`	Type
`Type Pos,_`	Typeset

At last, the type constr for the terms of the Calculus of Constructions is just :

```
type constr == sorts oper term;;
```

The syntax of the operators is the following :

Prop	DOP0 (Sort (Prop Null))
Set	DOP0 (Sort (Prop Pos))
$\lambda x : A.B$	DOP2 (Lambda, A, DLAM(Name x, B))
$(x : A)B$	DOP2 (Prod, A, DLAM(Name x, B))
$(f\ x_1\ \ldots\ x_n)$	DOPN (AppL, [| f; x_1;...; x_n |])

Notice that AppL is always done via DOPN, even if the application is only binary (so (M N) is represented by DOPN(AppL,[|M;N|])).

Examples.

- $[x : \mathsf{Set}]x$ is represented as

```
DOP2 (Lambda, DOP0 (Sort (Prop Pos)), DLAM (Name #x, Rel 1))
```

- $[P : \mathsf{Set} \to \mathsf{Prop}](x : \mathsf{Set})(Px)$ is represented as

```
DOP2 (Lambda, DOP2 (Prod, DOP0 (Sort (Prop Pos)),
                    DLAM (Anonymous, DOP0 (Sort (Prop Null)))),
     DLAM (Name #P, DOP2 (Prod, DOP0 (Sort (Prop Pos)),
                  DLAM (Name #x, DOPN (AppL, [|Rel 2; Rel 1|])))))
```

Constants. The case of constants is more complicated. The constants are stored into a table and referred to with a *section-path*. The *section-path* is a global name system to refer to any object without ambiguity. It can be seen as a filename, in which the sections are the directories. The type of *section-paths* is section_path (defined in src/meta/names.ml). It's a record of a "directory" (the list of the crossed sections), a "basename" (the identifier for the object) and a "kind" (CCI for the terms of the Calculus of Constructions, FW for the the terms of F_ω and OBJ for other objects). Here is such a path (pretty printed with string_of_path) :

$$\underbrace{\texttt{\#foo\#bar\#hat}}_{\text{dirpath}}\underbrace{\texttt{\#constantname}}_{\text{basename}}\underbrace{\texttt{.cci}}_{\text{kind}}$$

and it could correspond to a definition of the form :

```
Section foo.
Section bar.
Section hat.
Definition constantname := ... .
```

Once you close a section, say `hat` here, the discharge mechanism creates a new constant with an updated section-path (and keeps the old one in the closed section, so it is now unreachable). In our example, the new section-path for the constant `constantname` becomes :

```
#foo#bar#constantname.cci
```

The other part of a constant term is the environment of its definition. For instance, in the following definition :

```
Coq < Section foo.

Coq <   Variable A:Prop.

Coq <   Definition f := [x:Prop->Prop](x A).
```

the constant `f` depends on the variable `A` and this information is kept (when closing the section `foo`, we have to remember that `f` depends on `A`, and to do the corresponding abstraction). So a constant is a term of the form :

```
DOPN ( Const(sp) , l )
```

where `sp` is the section-path and `l` is the piece of the current environment needed for the definition of the constant. In our previous example, the corresponding term is :

```
DOPN ( Const #foo#f.cci , [| VAR #A |] )
```

and after having closed the section `foo`, it would become :

```
DOPN ( Const #f.cci , [| |] )
```

`f` being now equal to `[A:Prop][x:Prop->Prop](x A)`.

You can access the value or the type of a constant through the functions :

```
value const_value : readable_constraints -> constr -> constr;;
value const_type  : readable_constraints -> constr -> constr;;
```

where the first argument is the context of existential variables (associated to proof trees) and the second one a term of the form `DOP2(Const _,_)` (otherwise you get an exception `Match_failure`). The empty context for existential variables is `mt_evc` (`src/proofs/proof.ml`). Remember that constants are separated between transparent and opaque constants. Trying to get the value of an opaque constant would raise the exception `Failure "opaque"`.

Casts. One particular operator is the `Cast` operator. To "cast" a term means to give explicitly its type, as an information. So, the corresponding term is :

`DOP2( Cast ,` c `,` T `)`

where c is a term and T its type. Notice that :

- The pretty-printer *always* ignores casts, but that is changeable by setting the boolean reference `printer__print_casts` to `true`.
- Any cast in a term is verified by the type-checker, so they can be used to add information about the term which the system could infer, but which the programmer wants to declare.

Other operators. There are also other operators, for inductive types (`MutInd` and `MutConstruct`), for meta-variables (`Meta`), fix-points operators (`Fix`), ... We won't give details on those operators, which may differ with versions of the system, but their meaning (and sometimes their use) are not really difficult to understand.

11.1.2 Basic operations on terms

Basic operations are in `src/meta/generic.ml`: lifting, substitution, occurrences, free variables, application, ... The main ones are :

`value subst1 : 'a term -> 'a term -> 'a term.`
: `(subst1 M c)` substitutes `M` for `Rel(1)` in `c`.

`value occur_var : identifier -> 'a term -> bool.`
: Returns true if the corresponding variable appears in the term.

`value eq_term : 'a term -> 'a term -> bool.`
: α-equality for terms (this function ignores print names, casts and the iteration of applications, that is `(M N O) == ((M N) O)` where the parentheses specify the `DOPN`'s).

`value dependent : 'a term -> 'a term -> bool.`
: Returns true if the first term is a subterm of the second (for `eq_term`).

`value subst_var : identifier -> 'a term -> 'a term.`
: `(subst_var id c)` substitutes the corresponding de Bruijn index to every occurrence of `VAR(id)` in `c`.

`value SAPP : 'a term -> 'a term -> 'a term.`
: `(SAPP M N)` applies `M` to `N`, assuming that `M` is of the form `DLAM(n,Q)`.

Operations on CC's terms lie in `src/constr/term.ml`. Some of them are:

`value strip_outer_cast : constr -> constr.`
: Removes the outer casts (don't forget to do it before doing matching on terms).

`value applist : constr * constr list -> constr.`
: Returns the application of the first component to the second.

`value produit : identifier -> constr -> constr -> constr.`
: `(produit id T c)` returns the product $(id : T)c$.

`value lambda : identifier -> constr -> constr -> constr.`
: `(lambda id T c)` returns the abstraction $\lambda id : T.c$.

`value eq_constr : constr * constr -> bool.`
: α-conversion (ignores print names and casts).

`value subst_term : constr -> constr -> constr.`
: `subst_term` *un-substitutes*, that is if `(subst_term c t)`$\rightarrow$`M`, then `M[t/1]`$\rightarrow$`c`. `subst_term` uses `eq_term` to find copies of `t` in `c`.

Reduction functions lie in `src/proofs/reduction.ml` and are of type :

```
type reduction_function == constr -> constr
```

They generally compute weak head normal form, that is they stop on abstractions, products, constants and sorts. Reduction is performed under casts, and head casts are removed (reduction called *cast*). Iterations of applications are reduced like this :

`((M N) L1 ... Ln)` $\longrightarrow$ `(M N L1 ... Ln)`

(reduction called *app*). All the standard reduction functions performs the reductions *cast* and *app*. The reduction function `whd_castapp` performs only these two reductions.

The main reduction and conversion functions are the following :

`value whd_beta : reduction_function.`
: β-reduction.

`value whd_betaiota : reduction_function.`
: $\beta\iota$-reduction.

`value strong : reduction_function -> reduction_function.`
: Takes a reduction function and returns the associated recursive reduction function.

`value under_casts : reduction_function -> reduction_function.`
: Takes a reduction function and returns the same one, but performing under outer casts. `under_casts` preserves the outermost casts; otherwise all the other reduction functions will erase outermost casts.

`value conv : readable_constraints -> constr -> constr -> bool.`
: Equality of terms with universe adjustment.

`value conv_x : readable_constraints -> constr -> constr -> bool.`
: Equality of terms without universe adjustment.

11.2 Writing your own tactics

11.2.1 What is a tactic ?

In Coq a tactic is a function of type :

```
type tactic == goal sigma -> (goal list sigma * validation)
```

That is, a tactic takes a goal g (an object of type `goal sigma`) and returns the list (possibly empty) of the generated subgoals $g_1, \ldots, g_n$, together with a validation v. This validation has type :

```
type validation == (proof list -> proof)
```

and has the following interpretation : given a list of proofs $\pi_1, \ldots, \pi_n$, where π_i is a proof of g_i, v applied to $\pi_1, \ldots, \pi_n$ returns a proof of g. Here proofs can be incomplete proofs; but if $\pi_1, \ldots, \pi_n$ are complete proofs then the validation applied to those proofs returns a complete proof of g. Assume that `gls` is the current goal (of type `goal sigma`). This goal is essentially :

- *a conclusion*, a term a type `constr`, obtained with `(pf_concl gls)`;
- *a local context of hypothesis*, of type `constr signature`. It is exactly the context printed by the `Show` command. Is is obtained with `(pf_hyps gls)`.

About signatures. The type `signature` is a generic type for environments with global names :

```
type 'a signature == identifier list * 'a list
```

The first list contains the names, and the second one their corresponding objects — we assume here that the two lists have the same length — which are referred to with global names, using the `VAR` constructor. All the necessary functions to deal with signatures are in `src/meta/names.ml` (`add_sign`, `lookup_sign`,...).

For instance, if you enter at the `Coq` top-level `Lemma foo : (A:Prop)A->A.` then `Intros.` the current goal is now :

```
Coq < Intros.
```

and its signature is :

```
[ #H ; #A ],
[ DOP2 (Cast, VAR #A, DOP0 (Sort (Prop Null))) ,
  DOP2 (Cast, DOP0 (Sort (Prop Null)),
                           DOP0 (Sort (Type (Null, ...)))) ]
```

which can be seen, after removing the casts, as :

A	$Prop$
H	`VAR` A

The function `initial_sign` (in `src/constr/vartab.ml`) returns the signature of current global variables.
There is a second kind of signature using *de Bruijn* indexes instead of global names :

```
type 'a db_signature == (name * 'a) list
```

where an object of type `name` is either `(Name` *id*`)` or `Anonymous`.
These two signatures are mixed together in the type `env` of environments :

```
type ('a,'b) env = ENVIRON of 'a signature * 'b db_signature
```

which is just a couple of a signature and a de Bruijn signature. All the functions on environments are in `src/meta/names.ml` (`add_glob`, `add_rel`, `lookup_glob`, `lookup_rel`,...). To use functions over environments on signatures, just transform your signature in an environment with the `GLOB` function (which has type `'b signature -> ('b,'a) env`). For instance, you will usually look for a variable of name *id* with :

`(global (GLOB(initial_sign())) ` *id*`)`

11.2.2 Basic tactics and tacticals

There are numerous tactics in the system, in particular those of the top-level. Most of them lie in `src/env/tactics.ml`, and we give here some of them ::

`value intro : tactic.`
: The introduction tactic. (There is also `intros`.)

`value intro_using : identifier -> tactic.`
: Introduction with explicit name. (See also `intros_with` and `intros_until`.)

`value red : tactic.`
: The `Red` tactic. (See also `red_hyp`.)

`value cut_tac : constr -> tactic.`
: The `Cut` tactic.

`value exact : constr -> tactic.`
: The `Exact` tactic.

There are also functions to compose tactics — the so-called tacticals — in order to build more complex tactics from elementary ones. These tacticals are defined in `src/proofs/refiner.ml` :

`value IDTAC : tactic.`
: The identity tactic (just does nothing).

`value ORELSE : tactic -> tactic -> tactic.`
: Tries the first tactic and, in case of failure, applies the second one.

`value THEN : tactic -> tactic -> tactic.`
: Applies the first tactic, then the second one to each generated subgoal.

`value THENS : tactic -> tactic list -> tactic.`
: Applies a tactic, and then applies each tactic of the tactic list to the corresponding generated subgoal.

`value THENL : tactic -> tactic -> tactic.`
: Applies the first tactic, and then applies the second one to the last generated subgoal.

`value REPEAT : tactic -> tactic.`
: Applies the tactic until it fails (The tactic is applied to the goal, and then to every produced goal, and so on.)

`value FIRST : tactic list -> tactic.`
: Tries the tactics one by one until one succeeds.

`value TRY : tactic -> tactic.`
: Tries the tactic and, in case of failure, applies the `IDTAC` tactic to the original goal.

`value DO : int -> tactic -> tactic.`
: Applies the tactic a given number of times.

`value FAILTAC : tactic.`
: The failing tactic. It raises a `UserError` exception.

11.2.3 Handling terms inside a tactic

Inside a tactic, that is with a variable `gls` of type `goal sigma`, the system provides functions to handle terms in the context of the corresponding proof. Here are some of them :

`value pf_concl : goal sigma -> constr.`
: Returns the conclusion of the goal.

`value pf_hyps : goal sigma -> constr signature.`
: Returns the local context of the goal.

`value pf_global : goal sigma -> identifier -> constr.`
: Returns the corresponding term to an identifier, looking first in the context of the goal, then in the global context.

`value pf_type_of : goal sigma -> constr -> constr.`
: Checks if the term is well-typed in the current context and, if so, returns its type.

`value pf_nf : goal sigma -> constr -> constr.`
: Returns the normal form of the term.

As a general rule, a function taking a goal (of type `goal sigma`) as argument has a name of the form `pf_`*function-name*. All these functions are in `src/proofs/tacmach.ml`.

We can also do more complex manipulations on terms. Suppose we want to know is a term `t` is a conjunction. One can write :

```
let is_conj t =
  let sp = path_of_string "#Prelude#and.cci" in
  match whd_betaiota t with
      DOPN(AppL , [| DOPN(Ind sp',[||]) ; _ ; _ |]) -> sp=sp'
    | _ -> false
;;
```

but this is a bit complicated.

That's the reason why the system provides a better way to handle terms. The idea is to define *patterns* to do pattern matching and destructuring on terms.

To define these patterns we first indicate which modules have to be loaded. For example, in our case, the `Basis.v` module :

```
let mmk = make_module_marker ["#Basis.obj"];;
```

then we define the patterns as terms with "holes" (indicated by `?`). For example, the pattern for conjunction is defined as :

```
let and_pattern = put_pat mmk "(and ? ?)";;
```

If we want now to test if a term `t` is a conjunction and, in this case, to get the two sides of this conjunction, we will typically write :

```
...
if matches gls t and_pattern then
  let [A;B] = dest_match gls t and_pattern
  in ...
```

where `gls` is the current goal. These functions are defined in files `src/tactics/pattern.ml` and `src/tactics/tactics1.ml`.

There also exist similar functions to do second-order matching, in `sopattern.ml`, `somatch.ml` and `tactics1.ml` (in the directory `src/tactics`). Second-order matching means you can give a pattern like :

```
"(x,y:?)(and (?)@[x] (?)@[y])"
```

which means that we want A, $\lambda x.P$ and $\lambda y.Q$ if we match the expression `(x,y:`A`)(and` P Q`)`, where P is an expression containing x but not y, and Q is containing y but not x. The corresponding functions are `somatches` and `dest_somatch`. An exception may be raised by `dest_somatch` if the expression does not match the pattern, is malformed or if the pattern contains unknown global variables .

11.3 Tactic registration

Once the tactic is written, we have to turn it operational. Two operations are necessary :

- We need to *register* the tactic in the tactics table, so as to make it known by the system;
- We need to define the *grammar*'s and *syntax*'s rule(s) for the tactic.

11.3.1 Adding the tactic in the tactics table

This first operation just follows the code which defines the tactic, and use the function `register_tactic` (defined in `src/proofs/refiner.ml`). The type of this function is :

```
value register_tactic : string
     -> (tactic_arg list -> tactic)
     -> (readable_constraints -> goal -> tactic_expression -> st_ppcmds)
     -> (tactic_arg list -> tactic);;
```

The first argument is the name with which the tactic is registered. The second is the function which associate to the arguments the corresponding tactic. The type `tactic_arg` is the type of tactic arguments, defined in `src/proofs/proof_trees.mli` :

```
type tactic_arg =
    COMMAND of ast
  | CONSTR of constr
  | IDENTIFIER of identifier
  | INTEGER of int
  | BINDING of BindOcc * ast
  | PATTERN of int list * ast
```

```
       | UNFOLD of int list * identifier
       | QUOTED_STRING of string
       | TACEXP of ast
```

The third argument defines a pretty printing for the tactic. This pretty printer is used to print the script of the proof, for example just after the `Save` command.

`register_tactic` returns the function which associate the tactic to the arguments (that's not the second argument, because we now use the name with which the tactic is registered and not the function defining it). In general, we ignore this result.

For instance, the `intros_with` tactic, which corresponds to the top-level command `Intros H1 .. Hn`, is registered in this way :

```
let vernac_intros_with =
let gentac =
    register_tactic "IntrosWith"
    (fun al -> intros_with (map (fun (IDENTIFIER id) -> id) al))
    (fun _ _ (_,al) ->
         [< 'S"Intros" ; 'SPC ;
             prlist_with_sep (fun () -> [< 'SPC >])
                             (fun (IDENTIFIER id) -> print_id id)
                             al >])
in fun ids -> gentac (map (fun id -> IDENTIFIER id) ids)
;;
```

However, there are also registration functions adapted to particular syntaxes. They are defined in `src/env/tacmach.ml`, and their types are explicit enough :

`value register_atomic_tactic : string -> tactic -> tactic.`
Register an atomic tactic (a tactic without argument).

`value register_comarg_tactic : string -> (command -> tactic)`
`-> (command -> tactic).`
Register a tactic which takes a command.

`value register_numarg_tactic : string -> (int -> tactic)`
`-> (int -> tactic).`
Register a tactic which takes a integer.

`value register_ident_tactic : string -> (identifier -> tactic)`
`-> (identifier -> tactic).`
Register a tactic which takes an identifier.

`value register_string_tactic : string -> (string -> tactic)`
`-> (string -> tactic).`
Register a tactic which takes a string.

One can look into `src/env/tacentries.ml` to see how the different `Coq` top-level tactics are registered.

Remark. With `register_tactic` it's impossible to register two tactics with the same name, so it's impossible to register a tactic twice, when re-loading ML files. For that purpose one must use `overwriting_register_tactic`, and the corresponding functions `overwriting_`... for particular cases. Of course, these functions are for debugging purposes only.

11.3.2 Adding grammar's and syntax's entries

The next operation is the creation of a Coq file in which :

- we declare the Caml Light modules needed by the tactic;
- we define the grammar's rule(s) for the tactic;
- we define the syntax's rule(s) for pretty printing.

The syntax is the following :

```
Declare ML Module "fileA" "fileB" ... "fileZ".

Grammar tactic simple_tactic :=
  [ "tactic_name" ... ] -> ... .

Syntax tactic rule_name (tactic_function ...) 0
  "tactic_name" ... .
```

`fileA.ml`,..., `fileZ.ml` stand here for the Caml Light modules that must be loaded (given in the right order).

The syntax for `Grammar` and `Syntax` is given in other sections. For an atomic tactic, we will write :

```
Grammar tactic simple_tactic :=
  [ "tactic_name" ] -> [ $0 = (tactic_function) ].

Syntax tactic Tactic_name (tactic_function) 0
  "tactic_name".
```

and for a tactic which takes an integer as argument, we will write :

```
Grammar tactic simple_tactic :=
  [ "tactic_name" numarg($n) ] -> [ $0 = (tactic_function $n) ].

Syntax tactic Tactic_name (tactic_function ($PRIM $n)) 0
  "tactic_name" <$n:"Int":*>.
```

11.4 A complete example

We are now in position to give a complete example. Let us write a tactic, called `Mytactic`, which takes the name of an hypothesis, say `H`, and a term, say `t`, and instantiates `H` with `t` if `H` is a universal hypothesis.

It means that we have a goal of the form

```
...
H : (x:T)P
...
====================
g
```

and we want to replace it by the following one :

```
...
H : P[t/x]
...
====================
g
```

11.4.1 The Caml Light part

The tactic function

Our tactic takes two arguments : the name of one hypothesis, of type `identifier` and a term, of type `command`. So, it will be of the form :

```
let mytactic id c gls =
...
```

where `gls` has type `goal sigma`.

The first thing to do is to get the hypothesis corresponding to `id` in the proof signature, with `lookup_sign`. If `id` is not an hypothesis, `lookup_sign` raises `Not_found` and we send an error message to the user :

```
...
  let tid = try snd (lookup_sign id (pf_hyps gls))
                with Not_found -> error "No such hypothesis"
  in ...
```

Next, we want to check if `id` is an universal hypothesis. For this purpose we can write a general function `is_universal` of type `goal sigma -> constr -> bool` which returns `true` if and only if its second argument is a universal quantification (inside a goal given as first argument). We can write it as :

```
let is_universal gls T =
  match whd_betadeltaiota (Project gls) T with
    DOP2(Prod,_,DLAM(_,B)) -> dependent (Rel 1) B
  | _ -> false
;;
```

Notice that we perform a reduction on `T` before looking at its form. We can now check if `tid` is a universal quantification and send, if necessary, the good error message :

```
...
  if not(is_universal tid) then
        error ((string_of_id id) ^ " is not a universal hypothesis")
  else ...
```

We know now that `id` is an hypothesis of the form `(x:A)P`. We must check that `c` is a term a type `A` to do the substitution of `x` by `c` in `P`. It means that we check if `A` and the type of `c` are convertible :

```
...
let (DOP2(Prod,A,(DLAM _ as B))) = whd_betadeltaiota (Project gls) tid in
       let t = (pf_constr_of_com gls c) in
       if not(pf_conv_x gls A (pf_type_of gls t)) then
                 error "Illegal application"
  else ...
```

At last, we write the tactic part. It's just a cut of `P[c/x]` followed by, for the first subgoal, an introduction of the new hypothesis (we must before clear the old one), and for the second one, an application of the `exact` tactic :

```
...
  ( (cut_tac (SAPP B t))
    THENS [(clear_hyp [id]) THEN (introduction id) ;
           exact (applist(VAR id,[t])) ]  ) gls
;;
```

The tactic registration

We can now register the tactic, with `register_tactic`. Remember that the two arguments are an `identifier` and a `command` :

```
let mytactic_tac = register_tactic "mytactic"
  (fun [IDENTIFIER id; COMMAND c] -> mytactic id c)
  (fun sigma goal (_,[IDENTIFIER id; COMMAND c]) ->
      [< 'S"Mytactic" ; 'SPC ; print_id id ; 'SPC ;
         'S"with" ; pr_com sigma goal c >])
;;
```

The Caml Light file `mytactic.ml`

```
(**** mytactic.ml ****************************************************)
#open "std";;
#open "pp";;
#open "stdpp";;
#open "names";;
#open "generic";;
#open "term";;
#open "reduction";;
#open "proof_trees";;
```

```
#open "tacmach";;
#open "tactics";;
#infix "THENS";;
#infix "THEN";;

let is_universal gls T =
  match whd_betadeltaiota (Project gls) T with
    DOP2(Prod,_,DLAM(_,B)) -> dependent (Rel 1) B
  | _ -> false
;;

let mytactic id c gls =
  let tid = try snd (lookup_sign id (pf_hyps gls))
            with Not_found -> error "No such hypothesis"
  in if not(is_universal gls tid) then
        error ((string_of_id id) ^ " is not a universal hypothesis")
  else
    let (DOP2(Prod,A,(DLAM _ as B))) = whd_betadeltaiota (Project gls) tid in
       let t = (pf_constr_of_com gls c) in
       if not(pf_conv_x gls A (pf_type_of gls t)) then
            error "Illegal application"
  else ( (cut_tac (SAPP B t))
         THENS [(clear_hyp [id]) THEN (introduction id) ;
                exact (applist(VAR id,[t])) ] ) gls
;;

let mytactic_tac = register_tactic "mytactic"
  (fun [IDENTIFIER id; COMMAND c] -> mytactic id c)
  (fun sigma goal (_,[IDENTIFIER id; COMMAND c]) ->
      [< 'S"Mytactic" ; 'SPC ; print_id id ; 'SPC ;
         'S"with" ; pr_com sigma goal c >])
;;
(*********************************************************************)
```

11.4.2 The Coq file Mytactic.v

In Mytactic.v, we declare the file mytactic.ml and we give the grammar and syntax rules for our tactic :

```
(*** Mytactic.v *****************************************************)
Declare ML Module "mytactic".

Grammar tactic simple_tactic :=
  [ "Mytactic" identarg($id) "with" comarg($c) ]
                             -> [$0 = (mytactic $id $c) ].
```

```
Syntax tactic mytactic (mytactic $id $c) 0
  "Mytactic " <$id:"namehyp":*> " with " <$c:"Command":*>.
(*******************************************************************)
```

11.4.3 Compiling

In order to compile both `mytactic.ml` and `Mytactic.v`, let us write a `Makefile` in `MYTACTIC` to do the job :

```
### Makefile ########################################################
COQTOP=... # put here the right directory
ARCH=... # put here the right architecture
ZLIBS=  -I $(COQTOP)/src/lib/stream-pp -I $(COQTOP)/src/lib/util \
        -I $(COQTOP)/src/meta  -I $(COQTOP)/src/constr \
        -I $(COQTOP)/src/proofs -I $(COQTOP)/src/env \
        -I $(COQTOP)/src/tactics -I $(COQTOP)/src/link

all: mytactic.zo Mytactic.vo

Mytactic.vo: Mytactic.v mytactic.zo
        $(COQTOP)/bin/$(ARCH)/coqc Mytactic

mytactic.zo: mytactic.ml
        camlc $(ZLIBS) -c mytactic.ml
####################################################################
```

11.4.4 Use of the tactic

Once the compiling is done, we can use the tactic in a Coq session.

```
% coqtop -I MYTACTIC
Welcome to Coq V5.10 - ...

Coq <
```

We import the file `Mytactic.v` with the command :

```
Coq < Require Mytactic.
[Reinterning specification Mytactic ...done]
[Loading ML file mytactic ...done]

Coq <
```

The tactic is now known, and we can use it :

```
Coq < Variable P:nat -> Prop.
P is assumed
```

```
Coq < Lemma easy : ((x:nat)(P x)) -> (P (S (S O))).
1 subgoal

  ============================
   ((x:nat)(P x))->(P (S (S O)))

easy < Intro.
1 subgoal

  H : (x:nat)(P x)
  ============================
   (P (S (S O)))

easy < Mytactic H with (S (S O)).
1 subgoal

  H : (P (S (S O)))
  ============================
   (P (S (S O)))
```

11.5 Some tools

11.5.1 Debugger

For the moment, we don't have good debugging tools. Actually, we have just the `trace` mechanism, with `trace` and `untrace`.

We can leave the `Coq` top-level with the command `Drop` :

```
Coq < Drop.
#
```

and we are now in the Caml Light top-level.

In order to open the main modules and to define pretty printers for most types, just include the file `tactics/include.ml`:

```
include "include";;
```

We come back to the `Coq` top-level with the command :

```
go();;
```

11.5.2 Other tools

Other tools to simplify tactics writing (automatic computation of files dependencies, creation of a `Makefile`, ...) are described in chapter 14.

Chapter 12

The Program Tactic

The facilities described in this chapter pertain to a special aspect of the Coq system: how to associate to a functional program, whose specifications are written in Gallina, a proof of its correctness.

This methodology is based on the Curry-Howard isomorphism between functional programs and constructive proofs. This isomorphism allows the synthesis of a functional program from the constructive part of its proof of correctness. That is, it is possible to analyze a Coq proof, to erase all its non-informative parts (roughly speaking, removing the parts pertaining to sort Prop, considered as comments, to keep only the parts pertaining to sort Set).

This *realizability interpretation* was defined by Christine Paulin-Mohring in her PhD dissertation, and implemented as a *program extraction* facility in previous versions of Coq by Benjamin Werner. However, the corresponding certified program development methodology was very awkward: the user had to understand very precisely the extraction mechanism in order to guide the proof construction towards the desired algorithm. The facilities described in this chapter attempt to do the reverse: i.e. to try and generate the proof of correctness from the program itself, given as argument to a specialized tactic. This work is based on the PhD dissertation of Catherine Parent [61].

12.1 Developing certified programs: 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 developed 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.

12.2 Syntax for tactics

The user has to give two things : the specification (given as usual by a goal) and the program (see section 12.3). 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.

12.2.1 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 section 12.3 and it is associated to the current goal.

12.2.2 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.

12.2.3 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 no program is associated to the current goal, the `Program` tactic 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.

12.2.4 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 tries the `Auto` tactic on the logical subgoals. Note that the work of the `Program` tactic is considered to be 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.

12.2.5 Program_Expand

The `Program_Expand` tactic transforms the current program into the same program with the head constant expanded. This tactic particularly allows the user to force a program to be reduced before each application of the `Program` tactic.

12.3 Syntax for programs

12.3.1 Pure programs

The language to express programs is called Real*. Programs are explicitly 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.

*It corresponds to F_ω plus inductive definitions

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

12.3.2 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 `}` and preceding them with `:: ::`. 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.

12.3.3 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 section 12.4) which must always begin with an abstraction [x:A] where A is the type of the arguments of the function (also on which the ordering relation acts).

12.3.4 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`

and

`<P>if B then Q else R` abbreviates matching on boolean expressions, that is to say it abbreviates `<P>Match B with Q R`.

Moreover, a synthesis of implicit arguments has been added in order to allow the user to write a minimum of types in a program. Then, it is possible not to write a type inside a program term. This type has then to be automatically synthesized. For this, it is necessary to indicate where the implicit type to be synthesized appears. The syntax is the current one of implicit arguments in `Coq` : the question mark `?`.

This synthesis of implicit arguments is not possible everywhere in a program. In fact, the synthesis is only available inside a `Match`, a `Case` or a `Fix` construction (where `Fix` is a syntax for defining fixpoints).

Then, two macros have been introduced to suppress some question marks :

`<P>let (p,q:?)=Q in S` can be abbreviate into `<P>let (p,q)=Q in S` and `[x,y:?]T` can be abbreviate into `[x,y]T`.

12.3.5 Grammar

The grammar for programs is the following (see the section 2.2 for more explanation) :

pg::= ident | ?
| [x:pg]pg
| [x]pg
| (x:pg)pg
| pg->pg
| (pg pg … pg)
| <pg>Match pg with pg-list end
| <pg>Case pg with pg-list end
| Fix ident {ident/num : pg := pg with … with ident/num : pg := pg}
| pg :: :: { coqterm }
| [{x:coqterm}]pg
| <pg>let (x_1^1,…,$x_1^{k_1}$:A_1;…;x_n^1,…,$x_n^{k_n}$:A_n) = pg in pg
| <pg>let (x_1^1,…,$x_1^{k_1}$,…,x_n^1,…,$x_n^{k_n}$) = pg in pg
| <pg>if pg then pg else pg
| <pg>rec ident :: :: { 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.

12.4 Examples

12.4.1 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 :

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

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 Ack : nat->nat->nat->Prop :=
Coq <      AckO : (n:nat)(Ack O n (S n))
Coq <    | AcknO : (n,p:nat)(Ack n (S O) p)->(Ack (S n) O p)
Coq <    | AckSS : (n,m,p,q:nat)(Ack (S n) m q)->(Ack n q p)
Coq <                     ->(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 can be defined as a fixpoint :

```
Coq < Fixpoint ack_func [n:nat] : nat -> nat :=
Coq <   <nat->nat>Case n of
Coq <      (* 0 *) [m:nat](S m)
Coq <  (* (S n) *) [n':nat]
Coq <                   Fix ack_func2 {ack_func2/1 : nat -> nat :=
Coq <                       [m:nat]<nat>Case m of
Coq <                 (* 0 *) (ack_func n' (S O))
Coq <               (* S m *) [m':nat](ack_func n' (ack_func2 m'))
Coq <                            end}
Coq <     end.
```

The program is associated by using `Realizer ack_func`. The program is automatically expanded. Each realizer which is a constant is automatically expanded. Then, by repeating the `Program` tactic, three logical lemmas are generated and are easily solved by using the property Ack0, Ackn0 and AckSS.

```
Coq < Repeat Program.
```

12.4.2 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, ie, $\exists q \exists r.\ (a = b * q + r \wedge\ b > r)$:

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

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 < Realizer [n:nat]<nat->bool> Match n with
Coq <                        (* O *) [m]true
Coq <                        (* S *) [n',H,m]<bool> Case m of
Coq <                                 (* O *) false
Coq <                                 (* S *) [m'](H m')
Coq <                                end
Coq <                        end.
```

```
Coq < Program_all.

Coq < Save.
```

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

```
Coq < Realizer
Coq <       [b:nat](<nat*nat>rec div :: :: { lt }
Coq <               [a:nat]<nat*nat>if (le_gt_dec b a)
Coq <                         then <nat*nat>let (q,r) = (div (minus a b))
Coq <                                       in ((S q),r)
Coq <                         else (O,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 tactics `Program_all` or `Program` can be used, and the following logical lemmas are obtained :

```
Coq < Repeat Program.
```

The subgoals 4, 5 and 6 are resolved by `Auto` (if you use `Program_all` they don't appear, because `Program_all` tries to apply `Auto`). The other ones have to be solved by the user.

12.4.3 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 < Fixpoint inf_dec [n:nat] : nat -> bool :=
Coq < [m:nat]<bool>Case n of
Coq <                 true
Coq <                 [n':nat]<bool>Case m of
```

```
Coq <                                    false
Coq <                                    [m':nat](inf_dec n' m')
Coq <                             end
Coq <                    end.
```

The definition of the type `list` :

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

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 In [x:nat] : list->Prop
Coq <         := Inl  : (a:nat)(l:list)(In x l) -> (In x (cons a l))
Coq <         | Ineq : (l:list)(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 conjunction.

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

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

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

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

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

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

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) :

```
Coq < Program_all.
```

12.4.4 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) = splitting a m in
                 let m1 = quicksort l1 and
                 let m2 = quicksort l2 in m1@[a]@m2
```

Where splitting is defined by :

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

Suppose we give the following definitions in Coq :

Declaration of the ordering relation :

```
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 :

```
Coq < Fixpoint app [l:list] : list -> list
Coq <       := [m:list]<list>Case l of
Coq <                    (* nil *) m
Coq <             (* cons a l1 *) [a:A][l1:list](cons a (app l1 m)) end.
```

Definition of the permutation of two lists :

```
Coq < Inductive permut : list->list->Prop :=
Coq <     permut_nil  : (permut nil nil)
Coq <    |permut_tran : (l,m,n:list)(permut l m)->(permut m n)->(permut l n)
Coq <    |permut_cmil : (a:A)(l,m,n:list)
Coq <          (permut l (app m n))->(permut (cons a l) (mil a m n))
Coq <    |permut_milc : (a:A)(l,m,n:list)
Coq <          (permut (app m n) l)->(permut (mil a m n) (cons a l)).
```

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 < Section Rlist_.

Coq < Variable R : A->Prop.

Coq < Inductive Rlist  : list -> Prop :=
Coq <     Rnil : (Rlist nil)
Coq <   | Rcons : (x:A)(l:list)(R x)->(Rlist l)->(Rlist (cons x l)).


Coq < End Rlist_.

Coq < Hint Rnil Rcons.


Coq < Section Inf_Sup.

Coq < Hypothesis x : A.

Coq < Hypothesis l : list.

Coq < Definition inf_list := (Rlist (inf x) l).

Coq < Definition sup_list := (Rlist (sup x) l).

Coq < End Inf_Sup.
```

Definition of the property of a list to be sorted :

```
Coq < Inductive sort : list->Prop :=
Coq <      sort_nil : (sort nil)
Coq <    | sort_mil : (a:A)(l,m:list)(sup_list a l)->(inf_list a m)
Coq <         ->(sort l)->(sort m)->(sort (mil a l m)).
```

Then the goal to prove is $\forall l\ \exists m\ (sort\ m) \wedge (permut\ l\ m)$ and the specification is
`(l:list){m:list|(sort m)&(permut l m)`.

Let us first prove a preliminary lemma. Let us define `ltl` a well-founded ordering relation.

```
Coq < Definition ltl := [l,m:list](gt (length m) (length l)).
```

Let us then give a definition of `Splitting_spec` corresponding to
$\exists l_1 \exists l_2.\ (sup\_list\ a\ l_1) \wedge (inf\_list\ a\ l_2) \wedge (l \equiv l_1 @ l_2) \wedge (ltl\ l_1\ (a :: l)) \wedge (ltl\ l2\ (a :: l))$ and a theorem on this definition.

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

```
Coq < Theorem Splitting : (a:A)(l:list)(Splitting_spec a l).
```

```
Coq < Realizer [a:A][l:list]
Coq <     <list*list>Match l with
Coq <     (* nil *) (nil,nil)
Coq <   (* cons  *) [b,m,ll]<list*list>let (l1,l2) = ll in
Coq <                         <list*list>if (inf_sup a b)
Coq <                                    then (* inf a b *) (l1,(cons b l2))
Coq <                                    else (* sup a b *) ((cons b l1),l2)
Coq <     end.
```

```
Coq < Program_all.
```

```
Coq < Simpl; Auto.
```

```
Coq < Save.
```

The associated Real program to the specification we wanted to first prove and corresponding to the ML program will be :

```
Coq < Lemma Quicksort: (l:list){m:list|(sort m)&(permut l m)}.
```

```
Coq < Realizer <list>rec quick :: :: { ltl }
Coq <            [l:list]<list>Case l of
Coq <             (* nil *)  nil
Coq <            (* cons *)  [a,m]<list>let (l1,l2) = (Splitting a m) in
Coq <                                  (mil a (quick l1) (quick l2))
Coq <            end.
```

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

```
Coq < Program_all.
```

Chapter 13

The Coq commands

There are three Coq commands : two for interactive mode, and one for batch compilation. The options are (basically) the same for the three commands, and roughly described below. You can also look at the `man` pages of `coqtop`, `coq` and `coqc` for more details.

13.1 Interactive proof (coqtop, coq)

In the interactive mode, the user can develop his theories and proofs step by step in the Coq toplevel. The Coq toplevel is run by the command `coqtop` (which was formerly `coql`). This toplevel is based on a Caml light toplevel (to allow the dynamic link of tactics). You can switch to the Caml light toplevel with the command `Drop.`, and come back to the Coq toplevel with the command `go();;`. The command `coq` runs a stand-alone Coq toplevel, in which you cannot load tactics.

13.2 Batch compilation (coqc)

The `coqc` command takes a name *file* as argument. Then it looks for a vernacular file named *file*`.v`, and tries to compile it into a *file*`.vo` file (See 5.3).

13.3 Resource file

When Coq is launched, with either `coqtop`, `coq` or `coqc`, the resource file `$HOME/.coqrc.5.10` is loaded, where `$HOME` stands for your home directory. This file may contain, for instance, `AddPath` commands to add directories to the load path of Coq. You can use the environment variables `$COQTOP` and `$COQTH` to specify such directories (`$COQTOP` is the root directory of Coq and `$COQTH` is `$COQTOP/theories`, although you specified it with the `-theories` option). The default load path contains the following directories :

```
$COQTOP/tactics
$COQTOP/theories/ARITH
$COQTOP/theories/INIT
```

If there is no `.coqrc.5.10` file in your home directory, `Coq` will look for a `.coqrc` file instead. You can also specify an arbitrary name for the resource file (see option `-init-file` below), or the name of another user to load the resource file of someone else (see option `-user`).

It is possible to skip the loading of the resource file with the `-q` option.

13.4 Options

The following command-line options are recognized by the commands `coqc`, `coqtop`, and `coq`, except some of them which are not recognized by `coq`. See the manual pages for more details.

`-I` *directory*
: Add *directory* to the searched directories when looking for a file.

`-include` *directory*
: Identical to `-I` *directory*.

`-theories` *directory*
: Cause `Coq` to look for standard theories in *directory*`/ARITH` and *directory*`/INIT`.

`-is` *file*
: Cause `Coq` to use the state put in the file *file* as its input state. Mainly useful to build the standard input state.

`-inpustate` *file*
: Identical to `-is` *file*.

`-nois`
: Cause `Coq` to begin with an empty state. Mainly useful to build the standard input state.

`-notactics`
: Forbid the dynamic loading of tactics.

`-init-file` *file*
: Take *file* as rcfile, instead of `$HOME/.coqrc.5.10` file.

`-q`
: Cause `Coq` not to load your `$HOME/.coqrc.5.10` file.

`-user` *string*
: Take `~`*string*`/.coqrc.5.10` as rcfile, instead of `$HOME/.coqrc.5.10`

`-load-ml-source` *file*
: Load the Caml Light file *file*`.ml`

`-load-ml-object` *file*
: Load the Caml Light object file *file*`.zo`

`-load-vernac-source` *file*
: Load `Coq` file *file*`.v`

`-load-vernac-object` *file*
: Load `Coq` compiled file *file*`.vo`

`-require` *file*
: Loads `Coq` compiled file *file*`.vo` and import it (`Require` *file*).

`-batch`
: Batch mode : exit just after arguments parsing. This option is only used in the script `coqc`.

`-debug`
: Switch on the debug flag.

`-hash-cons`
: Switch on hash consing.

`-image` *file*
: This option sets the binary image to be used to be *file* instead of the standard one. Not of general use.

`-O` *module-set*
: Specify wich set of standard Caml Light modules will be used by `Coq`. See the Caml Light Reference Manual for details.

`-open` *module-set*
: Identical to `-O` *module-set*.

Chapter 14

Utilities

The distribution provides utilities to simplify some tedious works beside proof development, tactics writing or documentation.

14.1 Modules dependencies

In order to compute modules dependencies (so to use `make`), `Coq` comes with an appropriate tool, `coqdep`.

`coqdep` computes inter-module dependencies for `Coq` and Caml Light programs, and prints the dependencies on the standard output in a format readable by make. When a directory is given as argument, it is recursively looked at.

Dependencies of `Coq` modules are computed by looking at `Require` commands (`Require`, `Require Export`, `Require Import`, `Require Implementation`), and `Declare ML Module` commands.

Dependencies of Caml Light modules are computed by looking at `#open` directives and the double underscore notation *module__value*.

See the man page of `coqdep` for more details and options.

14.2 `Makefile`

When a proof development becomes large and is split into several files, it becomes crucial to use a tool like `make` to compile `Coq` modules.

The writing of a generic and complete `Makefile` may seem tedious and that's why `Coq` provides a tool to automate its creation, `do_Makefile`. Given the path to `$COQTOP` (the main directory of `Coq`) and the files to compile, `do_Makefile` prints a `Makefile` on the standard output. So one has just to run the command :

`do_Makefile` *Coq-path* $file_1$ ... $file_n$ `> Makefile`

The resulted `Makefile` has a target `depend` which computes the dependencies and adds them to the end of the `Makefile`. So each time you want to update the modules dependencies, type in :

`make depend`

There is also a target `all` to compile all the files $file_1$ ... $file_n$, and a generic target to produce a `.vo` file from the corresponding `.v` file (so you can do `make` *file*`.vo` to compile the file *file*`.v`).

14.3 Coq and LaTeX

When writing a documentation about a proof development, we provide a mechanical way to process Coq phrases embedded in LaTeX files : the `coq-tex` filter. This filter extracts Coq phrases embedded in LaTeX files, evaluates them, and insert the outcome of the evaluation after each phrase.

Starting with a file *file*`.tex` containing Coq phrases, the `coq-tex` filter produces a file *file*`.v.tex` with the Coq outcome. This LaTeX file must be compiled using the `coq` or `coq-sl` document style option (provided together with `coq-tex`).

See the man page of `coq-tex` for more details and options.

Remark. This Reference Manual and the Tutorial have been completely produced with `coq-tex`.

14.4 Coq and GNU Emacs

Coq comes with a Major mode for GNU Emacs, `coq.el`. This mode provides syntax highlighting (assuming your GNU Emacs library provides `hilit19.el`) and also a rudimentary indentation facility.

See the file `tools/emacs/README` for more details.

14.5 Module specification

Given a Coq vernacular file, the `gallina` filter extracts its specification (inductive types declarations, definitions, type of lemmas and theorems), removing the proofs parts of the file. The Coq file *fike*`.v` gives bearth to the specification file *file*`.g` (where the suffix `.g` stands for Gallina).

See the man page of `gallina` for more details and options.

14.6 Man pages

There are man pages for `coqtop`, `coq`, `coqc`, `coqdep`, `gallina` and `coq-tex`. Man pages are optionally installed (see installation instructions in file `INSTALL`, step 6).

Chapter 15

List of additional documentation

15.1 Tutorial

A companion volume to this reference manual, the Coq Tutorial, is aimed at gently introducing new users to developing proofs in Coq without assuming prior knowledge of type theory.

15.2 Installation Procedures

A INSTALL file in the distribution explains how to install Coq.

15.3 Incompatibilities with CoqV5.8

This short note describes known upward incompatibilities with the previous distribution CoqV5.8. It is contained in the document Incompatibilities.*.

15.4 Users friendly Recursive Definition

This document details some special syntax for recursive definitions. It is contained in the separate document Recursive-Definition.*.

15.5 CoInductive Types facilities

This document details some special syntax for coinductive definitions and associate tactics. It is contained in the separate document CoInductives.*.

15.6 Extraction facilities

This document details some special facilities to extract ML program files. It is contained in the separate document Extraction.*.

15.7 Anomalies

The separate document `Anomalies.*` gives a list of known anomalies and bugs of the system. Before communicating us an anomalous behavior, please check first whether it has been already reported in this document.

Bibliography

[1] Ph. Audebaud. Partial objects in the calculus of constructions. In *Proceedings of the sixth Conf. on Logic in Computer Science.* IEEE, 1991.

[2] Ph. Audebaud. *Extension du Calcul des Constructions par Points fixes.* PhD thesis, Université Bordeaux I, 1992.

[3] H. Barendregt. Lambda calculi with types. Technical Report 91-19, Catholic University Nijmegen, 1991. in Handbook of Logic in Computer Science, Vol II.

[4] H. Barendregt and T. Nipkow, editors. *Types for Proofs and Programs.* Springer-Verlag, 1994. LNCS 806.

[5] H.P. Barendregt. *The Lambda Calculus its Syntax and Semantics.* North-Holland, 1981.

[6] J.L. Bates and R.L. Constable. Proofs as programs. *ACM transactions on Programming Languages and Systems*, 7, 1985.

[7] M.J. Beeson. *Foundations of Constructive Mathematics, Metamathematical Studies.* Springer-Verlag, 1985.

[8] G. Bellin and J. Ketonen. A decision procedure revisited : Notes on direct logic, linear logic and its implementation. *Theoretical Computer Science*, 95:115–142, 1992.

[9] E. Bishop. *Foundations of Constructive Analysis.* McGraw-Hill, 1967.

[10] S. Boutin. Certification d'un compilateur ml en coq. Rapport de dea, Université Paris 7, septembre 1992.

[11] R. S. Boyer and J. S. Moore. *A computational logic.* ACM Monograph. Academic Press, 1979.

[12] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[13] Th. Coquand. *Une Théorie des Constructions.* PhD thesis, Université Paris 7, janvier 1985.

[14] Th. Coquand. An analysis of girard's paradox. In *Symposium on Logic in Computer Science*, Cambridge, MA, 1986. IEEE Computer Society Press.

[15] Th. Coquand. Metamathematical investigations of a Calculus of Constructions. In P. Oddifredi, editor, *Logic and Computer Science.* Academic Press, 1990. Rapport de recherche INRIA 1088, also in [36].

[16] Th. Coquand. Pattern matching with dependent types. In Nordstróm et al. [56].

[17] Th. Coquand. Infinite objects in type theory. In Barendregt and Nipkow [4]. LNCS 806.

[18] Th. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *EUROCAL'85*, Linz, 1985. Springer-Verlag. LNCS 203.

[19] Th. Coquand and G. Huet. Concepts mathématiques et informatiques formalisés dans le calcul des constructions. In The Paris Logic Group, editor, *Logic Colloquium'85*. North-Holland, 1987.

[20] Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.

[21] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*. Springer-Verlag, 1990. LNCS 417.

[22] N.J. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indag. Math.*, 34, 1972.

[23] N.J. de Bruijn. A survey of the project automath. In J.P. Seldin and J.R. Hindley, editors, *to H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[24] G. Dowek. Naming and scoping in a mathematical vernacular. Rapport de Recherche 1283, INRIA, 1990.

[25] G. Dowek. *Démonstration automatique dans le Calcul des Constructions*. PhD thesis, Université Paris 7, December 1991.

[26] G. Dowek. L'Indécidabilité du Filtrage du Troisième Ordre dans les Calculs avec Types Dépedants Constructeurs de Types. *Compte Rendu de l'Académie des Sciences*, I, 312(12):951–956, 1991. (The undecidability of Third Order Pattern Matching in Calculi with Dependent Types or Type Constructors).

[27] G. Dowek. A second order pattern matching algorithm in the cube of typed λ-calculi. In *Proceedings of Mathematical Foundation of Computer Science*, volume 520 of *Lecture Notes in Computer Science*, pages 151–160, 1991. Also Rapport de Recherche Inria.

[28] G. Dowek. The undecidability of pattern matching in calculi where primitive recursive functions are representable. To appear in Theoretical Computer Science, 1992.

[29] G. Dowek. Third order matching is decidable. *Annals of Pure and Applied Logic, 69, pp. 135-155*, 1994.

[30] G. Dowek. Lambda-calculus, combinators and the comprehension schema. In *Proceedings of the second international conference on typed lambda calculus and applications*, 1995.

[31] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq Proof Assistant User's Guide Version 5.8. Rapport Technique 154, INRIA, May 1993.

[32] P. Dybjer. Inductive sets and families in Martin-Löf's Type Theory and their set-theoretic semantics an inversion principle for Martin-Löf's type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, volume 14, pages 59–79. Cambridge University Press, 1991.

[33] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3), Sept 1992.

[34] J.-C. Filliâtre. Une procédure de décision pour le Calcul des Prédicats Direct. Etude et implémentation dans le système Coq. Master's thesis, DEA d'Informatique, ENS Lyon, Septembre 1994.

[35] Eric Fleury. Implantation des algorithmes de floyd et de dijkstra dans le Calcul des Constructions. Rapport de Stage, July 1990.

[36] Projet Formel. The calculus of constructions. documentation and user's guide, version 4.10. Rapport Technique 110, INRIA, 1989.

[37] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la téorie des types. In *Proceedings of the 2nd Scandinavian Logic Symposium*. North-Holland, 1970.

[38] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972.

[39] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989.

[40] D. Hirschkoff. Ecriture d'une tactique arithmétique pour le système coq. Master's thesis, DEA IARFA, Ecole des Ponts et Chaussées, Paris, Septembre 1994.

[41] W.A. Howard. The formulae-as-types notion of constructions. In J.P. Seldin and J.R. Hindley, editors, *to H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980. Unpublished 1969 Manuscript.

[42] G. Huet. Induction principles formalized in the Calculus of Constructions. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*. Elsevier Science, 1988. Also in Proceedings of TAPSOFT87, LNCS 249, Springer-Verlag, 1987, pp 276–286.

[43] G. Huet. The constructive engine. In R. Narasimhan, editor, *A perspective in Theoretical Computer Science. Commemorative Volume for Gift Siromoney*. World Scientific Publishing, 1989. also in [36].

[44] G. Huet, editor. *Logical Foundations of Functional Programming*. The UT Year of Programming Series. Addison-Wesley, 1989.

[45] G. Huet and G. Plotkin, editors. *Logical Frameworks*. Cambridge University Press, 1991.

[46] Gérard Huet. The gallina specification language : A case study. In *Proceedings of 12th FST/TCS Conference, New Delhi*. Springer Verlag LNCS 652, pp 229–240, 1992.

[47] Gérard Huet. Residual theory in λ-calculus: a formal development. *J. Functional Programming*, 4,3:371–394, 1994.

[48] Gérard Huet and Gordon Plotkin, editors. *Logical Environments*. Cambridge University Press, 1992.

[49] J. Ketonen and R. Weyhrauch. A decidable fragment of Predicate Calculus. *Theoretical Computer Science*, 32:297–307, 1984.

[50] S.C. Kleene. *Introduction to Metamathematics*. Bibliotheca Mathematica. North-Holland, 1952.

[51] J.-L. Krivine. *Lambda-calcul types et modèles*. études et recherche en informatique. Masson, 1990.

[52] F. Leclerc and C. Paulin-Mohring. Programming with streams in coq. a case study : The sieve of eratosthenes. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs, Types' 93*, volume 806 of *LNCS*. Springer-Verlag, 1994.

[53] P. Manoury and M. Simonot. Automatizing termination proof of recursively defined function. *TCS*, To appear.

[54] B. Nordström. Terminating general recursion. *BIT*, 28, 1988.

[55] B. Nordström, K. Peterson, and J. Smith. *Programming in Martin-Löf's Type Theory*. International Series of Monographs on Computer Science. Oxford Science Publications, 1990.

[56] B. Nordstróm, K. Petersson, and G. Plotkin, editors. *Proceedings of the 1992 Workshop on Types for Proofs and Programs*. Available by ftp at site ftp.inria.fr, 1992.

[57] C. Mu noz. Démonstration automatique dans la logique propositionnelle intuitionniste. Master's thesis, DEA d'Informatique Fondamentale, Université Paris 7, Septembre 1994.

[58] P. Odifreddi, editor. *Logic and Computer Science*. Academic Press, 1990.

[59] P. Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, 1984.

[60] C. Parent. Developing certified programs in the system Coq- The Program tactic. Technical Report 93-29, Ecole Normale Supérieure de Lyon, October 1993. Also in Proceedings of the BRA Workshop Types for Proofs and Programs, may 93.

[61] C. Parent. *Synthèse de preuves de programmes dans le calcul des constructions inductives*. Thèse de doctorat, Ecole Normale Supérieure de Lyon, 1995.

[62] M. Parigot. Recursive programming with proofs. *Theoretical Computer Science*, 94(2):335–356, 1992.

[63] M. Parigot, P. Manoury, and M. Simonot. Propre : A programming language with proofs. In A. Voronkov, editor, *Logic Programming and automated reasoning*, number 624 in LNCS, St. Petersburg, Russia, July 1992. Springer-Verlag.

[64] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM.

[65] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* PhD thesis, Université Paris 7, January 1989.

[66] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in LNCS, 1993. Also LIP research report 92-49, ENS Lyon.

[67] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[68] J. Rouyer. Développement de l'Algorithme d'Unification dans le Calcul des Constructions. To appear as a technical report, August 1992.

[69] A. Saïbi. Axiomatization of a lambda-calculus with explicit-substitutions in the coq system. Technical Report 2345, INRIA, Dec 1994.

[70] D. Terrasse. Traduction de TYPOL en COQ. Application à Mini ML. Rapport de dea, IARFA, September 1992.

[71] L. Théry, Y. Bertot, and G. Kahn. Real theorem provers deserve real user-interfaces. Rapport de Recherche 1684, INRIA Sophia, May 1992.

[72] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, an introduction.* Studies in Logic and the foundations of Mathematics, volumes 121 and 123. North-Holland, 1988.

[73] P. Weis and X. Leroy. *Le langage Caml.* InterEditions, 1993.

[74] B. Werner. *Une théorie des constructions inductives.* Thèse de doctorat, Université Paris 7, 1994.

Index

Unité de recherche Inria Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 Villers Lès Nancy
Unité de recherche Inria Rennes, Irisa, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Unité de recherche Inria Rhône-Alpes, 46 avenue Félix Viallet, 38031 Grenoble Cedex 1
Unité de recherche Inria Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 Le Chesnay Cedex
Unité de recherche Inria Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex

Éditeur

Inria, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)

ISSN 0249-6399