The Coq Proof Assistant

Reference Manual

August 5, 2009

Version 8.2pl1[1]

The Coq Development Team

TypiCal Project (formerly LogiCal)

[1]This research was partly supported by IST working group “Types”

V8.2pl1, August 5, 2009

Introduction

This document is the Reference Manual of version 8.2pl1 of the COQ proof assistant. A companion volume, the COQ Tutorial, is provided for the beginners. It is advised to read the Tutorial first. A book [14] on practical uses of the COQ system was published in 2004 and is a good support for both the beginner and the advanced user.

The COQ system is designed to develop mathematical proofs, and especially to write formal specifications, programs and to verify that programs are correct with respect to their specification. It provides a specification language named GALLINA. Terms of GALLINA can represent programs as well as properties of these programs and proofs of these properties. Using the so-called *Curry-Howard isomorphism*, programs, properties and proofs are formalized in the same language called *Calculus of Inductive Constructions*, that is a λ-calculus with a rich type system. All logical judgments in COQ are typing judgments. The very heart of the Coq system is the type-checking algorithm that checks the correctness of proofs, in other words that checks that a program complies to its specification. COQ also provides an interactive proof assistant to build proofs using specific programs called *tactics*.

All services of the COQ proof assistant are accessible by interpretation of a command language called *the vernacular*.

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.

- 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.

These two modes are documented in Chapter 13.

Other modes of interaction with COQ are possible: through an emacs shell window, an emacs generic user-interface for proof assistant (ProofGeneral [1]) or through a customized interface (PCoq [128]). These facilities are not documented here. There is also a COQ Integrated Development Environment described in Chapter 15.

How to read this book

This is a Reference Manual, not a User Manual, then it is not made for a continuous reading. However, it has some structure that is explained below.

- The first part describes the specification language, Gallina. Chapters 1 and 2 describe the concrete syntax as well as the meaning of programs, theorems and proofs in the Calculus of Inductive Constructions. Chapter 3 describes the standard library of Coq. Chapter 4 is a mathematical description of the formalism. Chapter 5 describes the module system.

- The second part describes the proof engine. It is divided in five chapters. Chapter 6 presents all commands (we call them *vernacular commands*) that are not directly related to interactive proving: requests to the environment, complete or partial evaluation, loading and compiling files. How to start and stop proofs, do multiple proofs in parallel is explained in Chapter 7. In Chapter 8, all commands that realize one or more steps of the proof are presented: we call them *tactics*. The language to combine these tactics into complex proof strategies is given in Chapter 9. Examples of tactics are described in Chapter 10.

- The third part describes how to extend the syntax of Coq. It corresponds to the Chapter 12.

- In the fourth part more practical tools are documented. First in Chapter 13, the usage of `coqc` (batch mode) and `coqtop` (interactive mode) with their options is described. Then, in Chapter 14, various utilities that come with the Coq distribution are presented. Finally, Chapter 15 describes the Coq integrated development environment.

At the end of the document, after the global index, the user can find specific indexes for tactics, vernacular commands, and error messages.

List of additional documentation

This manual does not contain all the documentation the user may need about Coq. Various informations can be found in the following documents:

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. In a second step, the user can read also the tutorial on recursive types (document `RecTutorial.ps`).

Addendum The fifth part (the Addendum) of the Reference Manual is distributed as a separate document. It contains more detailed documentation and examples about some specific aspects of the system that may interest only certain users. It shares the indexes, the page numbers and the bibliography with the Reference Manual. If you see in one of the indexes a page number that is outside the Reference Manual, it refers to the Addendum.

Installation A text file INSTALL that comes with the sources explains how to install Coq.

The Coq standard library A commented version of sources of the Coq standard library (including only the specifications, the proofs are removed) is given in the additional document `Library.ps`.

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_ω, 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 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

Credits: addendum for version 6.1

The present version 6.1 of COQ is based on the V5.10 architecture. It was ported to the new language Objective Caml by Bruno Barras. The underlying framework has slightly changed and allows more conversions between sorts.

The new version provides powerful tools for easier developments.

Cristina Cornes designed an extension of the COQ syntax to allow definition of terms using a powerful pattern-matching analysis in the style of ML programs.

Amokrane Saïbi wrote a mechanism to simulate inheritance between types families extending a proposal by Peter Aczel. He also developed a mechanism to automatically compute which arguments of a constant may be inferred by the system and consequently do not need to be explicitly written.

Yann Coscoy designed a command which explains a proof term using natural language. Pierre Crégut built a new tactic which solves problems in quantifier-free Presburger Arithmetic. Both functionalities have been integrated to the COQ system by Hugo Herbelin.

Samuel Boutin designed a tactic for simplification of commutative rings using a canonical set of rewriting rules and equality modulo associativity and commutativity.

Finally the organisation of the COQ distribution has been supervised by Jean-Christophe Filliâtre with the help of Judicaël Courant and Bruno Barras.

Lyon, Nov. 18th 1996
Christine Paulin

Credits: addendum for version 6.2

In version 6.2 of COQ, the parsing is done using camlp4, a preprocessor and pretty-printer for CAML designed by Daniel de Rauglaudre at INRIA. Daniel de Rauglaudre made the first adaptation of COQ for camlp4, this work was continued by Bruno Barras who also changed the structure of COQ abstract syntax trees and the primitives to manipulate them. The result of these changes is a faster parsing procedure with greatly improved syntax-error messages. The user-interface to introduce grammar or pretty-printing rules has also changed.

Eduardo Giménez redesigned the internal tactic libraries, giving uniform names to Caml functions corresponding to COQ tactic names.

Bruno Barras wrote new more efficient reductions functions.

Hugo Herbelin introduced more uniform notations in the COQ specification language: the definitions by fixpoints and pattern-matching have a more readable syntax. Patrick Loiseleur introduced user-friendly notations for arithmetic expressions.

New tactics were introduced: Eduardo Giménez improved a mechanism to introduce macros for tactics, and designed special tactics for (co)inductive definitions; Patrick Loiseleur designed a tactic to simplify polynomial expressions in an arbitrary commutative ring which generalizes the previous tactic implemented by Samuel Boutin. Jean-Christophe Filliâtre introduced a tactic for refining a goal, using a proof term with holes as a proof scheme.

David Delahaye designed the SearchIsos tool to search an object in the library given its type (up to isomorphism).

Henri Laulhère produced the COQ distribution for the Windows environment.

Finally, Hugo Herbelin was the main coordinator of the COQ documentation with principal contributions by Bruno Barras, David Delahaye, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin and Patrick Loiseleur.

Orsay, May 4th 1998
Christine Paulin

Credits: addendum for version 6.3

The main changes in version V6.3 was the introduction of a few new tactics and the extension of the guard condition for fixpoint definitions.

B. Barras extended the unification algorithm to complete partial terms and solved various tricky bugs related to universes.
D. Delahaye developed the `AutoRewrite` tactic. He also designed the new behavior of `Intro` and provided the tacticals `First` and `Solve`.
J.-C. Filliâtre developed the `Correctness` tactic.
E. Giménez extended the guard condition in fixpoints.
H. Herbelin designed the new syntax for definitions and extended the `Induction` tactic.
P. Loiseleur developed the `Quote` tactic and the new design of the `Auto` tactic, he also introduced the index of errors in the documentation.
C. Paulin wrote the `Focus` command and introduced the reduction functions in definitions, this last feature was proposed by J.-F. Monin from CNET Lannion.

Orsay, Dec. 1999
Christine Paulin

Credits: versions 7

The version V7 is a new implementation started in September 1999 by Jean-Christophe Filliâtre. This is a major revision with respect to the internal architecture of the system. The COQ version 7.0 was distributed in March 2001, version 7.1 in September 2001, version 7.2 in January 2002, version 7.3 in May 2002 and version 7.4 in February 2003.

Jean-Christophe Filliâtre designed the architecture of the new system, he introduced a new representation for environments and wrote a new kernel for type-checking terms. His approach was to use functional data-structures in order to get more sharing, to prepare the addition of modules and also to get closer to a certified kernel.

Hugo Herbelin introduced a new structure of terms with local definitions. He introduced "qualified" names, wrote a new pattern-matching compilation algorithm and designed a more compact algorithm for checking the logical consistency of universes. He contributed to the simplification of COQ internal structures and the optimisation of the system. He added basic tactics for forward reasoning and coercions in patterns.

David Delahaye introduced a new language for tactics. General tactics using pattern-matching on goals and context can directly be written from the COQ toplevel. He also provided primitives for the design of user-defined tactics in CAML.

Micaela Mayero contributed the library on real numbers. Olivier Desmettre extended this library with axiomatic trigonometric functions, square, square roots, finite sums, Chasles property and basic plane geometry.

Jean-Christophe Filliâtre and Pierre Letouzey redesigned a new extraction procedure from COQ terms to CAML or HASKELL programs. This new extraction procedure, unlike the one implemented in previous version of COQ is able to handle all terms in the Calculus of Inductive Constructions, even involving universes and strong elimination. P. Letouzey adapted user contributions to extract ML programs when it was sensible. Jean-Christophe Filliâtre wrote `coqdoc`, a documentation tool for COQ libraries usable from version 7.2.

Bruno Barras improved the reduction algorithms efficiency and the confidence level in the correctness of COQ critical type-checking algorithm.

Yves Bertot designed the `SearchPattern` and `SearchRewrite` tools and the support for the PCOQ interface (http://www-sop.inria.fr/lemme/pcoq/).

Micaela Mayero and David Delahaye introduced `Field`, a decision tactic for commutative fields.

Christine Paulin changed the elimination rules for empty and singleton propositional inductive types.

Loïc Pottier developed `Fourier`, a tactic solving linear inequalities on real numbers.

Pierre Crégut developed a new version based on reflexion of the `Omega` decision tactic.

Claudio Sacerdoti Coen designed an XML output for the COQ modules to be used in the Hypertextual Electronic Library of Mathematics (HELM cf http://www.cs.unibo.it/helm).

A library for efficient representation of finite maps using binary trees contributed by Jean Goubault was integrated in the basic theories.

Pierre Courtieu developed a command and a tactic to reason on the inductive structure of recursively defined functions.

Jacek Chrząszcz designed and implemented the module system of COQ whose foundations are in Judicaël Courant's PhD thesis.

The development was coordinated by C. Paulin.

Many discussions within the Démons team and the LogiCal project influenced significantly the design of COQ especially with J. Courant, J. Duprat, J. Goubault, A. Miquel, C. Marché, B. Monate and B. Werner.

Intensive users suggested improvements of the system : Y. Bertot, L. Pottier, L. Théry, P. Zimmerman from INRIA, C. Alvarado, P. Crégut, J.-F. Monin from France Telecom R & D.

Orsay, May. 2002
Hugo Herbelin & Christine Paulin

Credits: version 8.0

COQ version 8 is a major revision of the COQ proof assistant. First, the underlying logic is slightly different. The so-called *impredicativity* of the sort `Set` has been dropped. The main reason is that it is inconsistent with the principle of description which is quite a useful principle for formalizing mathematics within classical logic. Moreover, even in an constructive setting, the impredicativity of `Set` does not add so much in practice and is even subject of criticism from a large part of the intuitionistic mathematician community. Nevertheless, the impredicativity of `Set` remains optional for users interested in investigating mathematical developments which rely on it.

Secondly, the concrete syntax of terms has been completely revised. The main motivations were

- a more uniform, purified style: all constructions are now lowercase, with a functional programming perfume (e.g. abstraction is now written `fun`), and more directly accessible to the novice (e.g. dependent product is now written `forall` and allows omission of types). Also, parentheses and are no longer mandatory for function application.

- extensibility: some standard notations (e.g. "<" and ">") were incompatible with the previous syntax. Now all standard arithmetic notations (=, +, *, /, <, <=, ... and more) are directly part of the syntax.

Together with the revision of the concrete syntax, a new mechanism of *interpretation scopes* permits to reuse the same symbols (typically +, -, *, /, <, <=) in various mathematical theories without any ambiguities for Coq, leading to a largely improved readability of Coq scripts. New commands to easily add new symbols are also provided.

Coming with the new syntax of terms, a slight reform of the tactic language and of the language of commands has been carried out. The purpose here is a better uniformity making the tactics and commands easier to use and to remember.

Thirdly, a restructuration and uniformisation of the standard library of Coq has been performed. There is now just one Leibniz' equality usable for all the different kinds of Coq objects. Also, the set of real numbers now lies at the same level as the sets of natural and integer numbers. Finally, the names of the standard properties of numbers now follow a standard pattern and the symbolic notations for the standard definitions as well.

The fourth point is the release of CoqIDE, a new graphical gtk2-based interface fully integrated to Coq. Close in style from the Proof General Emacs interface, it is faster and its integration with Coq makes interactive developments more friendly. All mathematical Unicode symbols are usable within CoqIDE.

Finally, the module system of Coq completes the picture of Coq version 8.0. Though released with an experimental status in the previous version 7.4, it should be considered as a salient feature of the new version.

Besides, Coq comes with its load of novelties and improvements: new or improved tactics (including a new tactic for solving first-order statements), new management commands, extended libraries.

Bruno Barras and Hugo Herbelin have been the main contributors of the reflexion and the implementation of the new syntax. The smart automatic translator from old to new syntax released with Coq is also their work with contributions by Olivier Desmettre.

Hugo Herbelin is the main designer and implementor of the notion of interpretation scopes and of the commands for easily adding new notations.

Hugo Herbelin is the main implementor of the restructuration of the standard library.

Pierre Corbineau is the main designer and implementor of the new tactic for solving first-order statements in presence of inductive types. He is also the maintainer of the non-domain specific automation tactics.

Benjamin Monate is the developer of the CoqIDE graphical interface with contributions by Jean-Christophe Filliâtre, Pierre Letouzey, Claude Marché and Bruno Barras.

Claude Marché coordinated the edition of the Reference Manual for Coq V8.0.

Pierre Letouzey and Jacek Chrząszcz respectively maintained the extraction tool and module system of Coq.

Jean-Christophe Filliâtre, Pierre Letouzey, Hugo Herbelin and contributors from Sophia-Antipolis and Nijmegen participated to the extension of the library.

Julien Narboux built a NSIS-based automatic Coq installation tool for the Windows platform.

Hugo Herbelin and Christine Paulin coordinated the development which was under the responsability of Christine Paulin.

Palaiseau & Orsay, Apr. 2004
Hugo Herbelin & Christine Paulin
(updated Apr. 2006)

Credits: version 8.1

COQ version 8.1 adds various new functionalities.

Benjamin Grégoire implemented an alternative algorithm to check the convertibility of terms in the COQ type-checker. This alternative algorithm works by compilation to an efficient bytecode that is interpreted in an abstract machine similar to Xavier Leroy's ZINC machine. Convertibility is performed by comparing the normal forms. This alternative algorithm is specifically interesting for proofs by reflection. More generally, it is convenient in case of intensive computations.

Christine Paulin implemented an extension of inductive types allowing recursively non uniform parameters. Hugo Herbelin implemented sort-polymorphism for inductive types.

Claudio Sacerdoti Coen improved the tactics for rewriting on arbitrary compatible equivalence relations. He also generalized rewriting to arbitrary transition systems.

Claudio Sacerdoti Coen added new features to the module system.

Benjamin Grégoire, Assia Mahboubi and Bruno Barras developed a new more efficient and more general simplification algorithm on rings and semi-rings.

Laurent Théry and Bruno Barras developed a new significantly more efficient simplification algorithm on fields.

Hugo Herbelin, Pierre Letouzey, Julien Forest, Julien Narboux and Claudio Sacerdoti Coen added new tactic features.

Hugo Herbelin implemented matching on disjunctive patterns.

New mechanisms made easier the communication between COQ and external provers. Nicolas Ayache and Jean-Christophe Filliâtre implemented connections with the provers CVCL, SIMPLIFY and ZENON. Hugo Herbelin implemented an experimental protocol for calling external tools from the tactic language.

Matthieu Sozeau developed RUSSELL, an experimental language to specify the behavior of programs with subtypes.

A mechanism to automatically use some specific tactic to solve unresolved implicit has been implemented by Hugo Herbelin.

Laurent Théry's contribution on strings and Pierre Letouzey and Jean-Christophe Filliâtre's contribution on finite maps have been integrated to the COQ standard library. Pierre Letouzey developed a library about finite sets "à la Objective Caml". With Jean-Marc Notin, he extended the library on lists. Pierre Letouzey's contribution on rational numbers has been integrated and extended..

Pierre Corbineau extended his tactic for solving first-order statements. He wrote a reflection-based intuitionistic tautology solver.

Pierre Courtieu, Julien Forest and Yves Bertot added extra support to reason on the inductive structure of recursively defined functions.

Jean-Marc Notin significantly contributed to the general maintenance of the system. He also took care of coqdoc.

Pierre Castéran contributed to the documentation of (co-)inductive types and suggested improvements to the libraries.

Pierre Corbineau implemented the C-zar mathematical proof language, usable in combination with the tactic-based style of proof.

Finally, many users suggested improvements of the system through the Coq-Club mailing list and bug-tracker systems, especially user groups from INRIA Rocquencourt, Radbout University, University of Pennsylvania and Yale University.

Palaiseau, July 2006
Hugo Herbelin

Credits: version 8.2

COQ version 8.2 adds new features, new libraries and improves on many various aspects.

Regarding the language of Coq, the main novelty is the introduction by Matthieu Sozeau of a package of commands providing Haskell-style type classes. Type classes, that come with a few convenient features such as type-based resolution of implicit arguments, plays a new role of landmark in the architecture of Coq with respect to automatization. For instance, thanks to type classes support, Matthieu Sozeau could implement a new resolution-based version of the tactics dedicated to rewriting on arbitrary transitive relations.

Another major improvement of Coq 8.2 is the evolution of the arithmetic libraries and of the tools associated to them. Benjamin Grégoire and Laurent Théry contributed a modular library for building arbitrarily large integers from bounded integers while Evgeny Makarov contributed a modular library of abstract natural and integer arithmetics together with a few convenient tactics. On his side, Pierre Letouzey made numerous extensions to the arithmetic libraries on $\mathbb{Z}$ and $\mathbb{Q}$, including extra support for automatization in presence of various number-theory concepts.

Frédéric Besson contributed a reflexive tactic based on Krivine-Stengle Positivstellensatz (the easy way) for validating provability of systems of inequalities. The platform is flexible enough to support the validation of any algorithm able to produce a "certificate" for the Positivstellensatz and this covers the case of Fourier-Motzkin (for linear systems in $\mathbb{Q}$ and $\mathbb{R}$), Fourier-Motzkin with cutting planes (for linear systems in $\mathbb{Z}$) and sum-of-squares (for non-linear systems). Evgeny Makarov made the platform generic over arbitrary ordered rings.

Arnaud Spiwack developed a library of 31-bits machine integers and, relying on Benjamin Grégoire and Laurent Théry's library, delivered a library of unbounded integers in base 2^{31}. As importantly, he developed a notion of "retro-knowledge" so as to safely extend the kernel-located bytecode-based efficient evaluation algorithm of Coq version 8.1 to use 31-bits machine arithmetics for efficiently computing with the library of integers he developed.

Beside the libraries, various improvements contributed to provide a more comfortable end-user language and more expressive tactic language. Hugo Herbelin and Matthieu Sozeau improved the pattern-matching compilation algorithm (detection of impossible clauses in pattern-matching, automatic inference of the return type). Hugo Herbelin, Pierre Letouzey and Matthieu Sozeau contributed various new convenient syntactic constructs and new tactics or tactic features: more inference of redundant information, better unification, better support for proof or definition by fixpoint, more expressive rewriting tactics, better support for meta-variables, more convenient notations, ...

Élie Soubiran improved the module system, adding new features (such as an "include" command) and making it more flexible and more general. He and Pierre Letouzey improved the support for modules in the extraction mechanism.

Matthieu Sozeau extended the RUSSELL language, ending in an convenient way to write programs of given specifications, Pierre Corbineau extended the C-zar mathematical proof language and the automatization tools that accompany it and added its documentation to the Reference Manual, Pierre Letouzey supervised and extended various parts the standard library, Stéphane Glondu contributed a few tactics and improvements, Jean-Marc Notin provided help in debugging, general maintenance and `coqdoc` support, Vincent Siles contributed extensions of the `Scheme` command and of `injection`.

Bruno Barras implemented the `coqchk` tool: this is a stand-alone type-checker that can be used to certify `.vo` files. Especially, as this verifier runs in a separate process, it is granted not to be "hijacked" by virtually malicious extensions added to COQ.

Yves Bertot, Jean-Christophe Filliâtre, Pierre Courtieu and Julien Forest acted as maintainers of features they implemented in previous versions of Coq.

Julien Narboux contributed to CoqIDE. Nicolas Tabareau made the adaptation of the interface of the old "setoid rewrite" tactic to the new version. Lionel Mamane worked on the interaction between Coq and its external interfaces. With Samuel Mimram, he also helped making Coq compatible with recent software tools. Russell O'Connor, Cezary Kaliscyk, Milad Niqui contributed to improved the libraries of integers, rational, and real numbers. We also thank many users and partners for suggestions and feedback, in particular Pierre Castéran and Arthur Charguéraud, the INRIA Marelle team, Georges Gonthier and the INRIA-Microsoft Mathematical Components team, the Foundations group at Radbout university in Nijmegen, reporters of bugs and participants to the Coq-Club mailing list.

Palaiseau, June 2008
Hugo Herbelin

Table of contents

Part I

The language

Chapter 1

The GALLINA specification language

This chapter describes GALLINA, the specification language of COQ. It allows to develop mathematical theories and to prove specifications of programs. The theories are built from axioms, hypotheses, parameters, lemmas, theorems and definitions of constants, functions, predicates and sets. The syntax of logical objects involved in theories is described in Section 1.2. The language of commands, called *The Vernacular* is described in section 1.3.

In COQ, logical objects are typed to ensure their logical correctness. The rules implemented by the typing algorithm are described in Chapter 4.

About the grammars in the manual

Grammars are presented in Backus-Naur form (BNF). Terminal symbols are set in `typewriter font`. In addition, there are special notations for regular expressions.

An expression enclosed in square brackets *[...]* means at most one occurrence of this expression (this corresponds to an optional component).

The notation "*entry* `sep` ... `sep` *entry*" stands for a non empty sequence of expressions parsed by *entry* and separated by the literal "`sep`"[1].

Similarly, the notation "*entry* ... *entry*" stands for a non empty sequence of expressions parsed by the "*entry*" entry, without any separator between.

At the end, the notation "*[entry* `sep` ... `sep` *entry]*" stands for a possibly empty sequence of expressions parsed by the "*entry*" entry, separated by the literal "`sep`".

1.1 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. They can contain any character. However, string literals must be correctly closed. Comments are treated as blanks.

Identifiers and access identifiers Identifiers, written *ident*, are sequences of letters, digits, _ and `'`, that do not start with a digit or `'`. That is, they are recognized by the following lexical class:

[1] This is similar to the expression "*entry* { `sep` *entry* }" in standard BNF, or "*entry* (`sep` *entry*)*" in the syntax of regular expressions.

first_letter ::= `a..z|A..Z|_|unicode-letter`
subsequent_letter ::= `a..z|A..Z|0..9|_|'|unicode-letter|unicode-id-part`
ident ::= *first_letter* [*subsequent_letter*...*subsequent_letter*]

All characters are meaningful. In particular, identifiers are case-sensitive. The entry `unicode-letter` non-exhaustively includes Latin, Greek, Gothic, Cyrillic, Arabic, Hebrew, Georgian, Hangul, Hiragana and Katakana characters, CJK ideographs, mathematical letter-like symbols, hyphens, non-breaking space, ... The entry `unicode-id-part` non-exhaustively includes symbols for prime letters and subscripts.

Access identifiers, written *access_ident*, are identifiers prefixed by `.` (dot) without blank. They are used in the syntax of qualified identifiers.

Natural numbers and integers Numerals are sequences of digits. Integers are numerals optionally preceded by a minus sign.

digit ::= `0..9`
num ::= *digit*...*digit*
integer ::= *[-]num*

Strings Strings are delimited by `"` (double quote), and enclose a sequence of any characters different from `"` or the sequence `""` to denote the double quote character. In grammars, the entry for quoted strings is *string*.

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

```
_       as       at   cofix  else    end
exists  exists2  fix  for    forall  fun
if      IF       in   let    match   mod
Prop    return   Set  then   Type    using
where   with
```

Special tokens The following sequences of characters are special tokens:

```
!   %   &   &&  (    ()   )
*   +   ++  ,   -    ->   .
.(  ..  /   /\  :    ::   :<
:=  :>  ;   <   <-   <->  <:
<=  <>  =   =>  =_D  >    >->
>=  ?   ?=  @   [    \/   ]
^   {   |   |-  ||   }    ~
```

Lexical ambiguities are resolved according to the "longest match" rule: when a sequence of non alphanumerical 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.

1.2 Terms

1.2.1 Syntax of terms

Figures 1.1 and 1.2 describe the basic set of terms which form the *Calculus of Inductive Constructions* (also called pCIC). The formal presentation of pCIC is given in Chapter 4. Extensions of this syntax are given in chapter 2. How to customize the syntax is described in Chapter 12.

1.2.2 Types

COQ terms are typed. COQ types are recognized by the same syntactic class as *term*. We denote by *type* the semantic subclass of types inside the syntactic class *term*.

1.2.3 Qualified identifiers and simple identifiers

Qualified identifiers (*qualid*) denote *global constants* (definitions, lemmas, theorems, remarks or facts), *global variables* (parameters or axioms), *inductive types* or *constructors of inductive types*. *Simple identifiers* (or shortly *ident*) are a syntactic subset of qualified identifiers. Identifiers may also denote local *variables*, what qualified identifiers do not.

1.2.4 Numerals

Numerals have no definite semantics in the calculus. They are mere notations that can be bound to objects through the notation mechanism (see Chapter 12 for details). Initially, numerals are bound to Peano's representation of natural numbers (see 3.1.3).

Note: negative integers are not at the same level as *num*, for this would make precedence unnatural.

1.2.5 Sorts

There are three sorts Set, Prop and Type.

- Prop is the universe of *logical propositions*. The logical propositions themselves are typing the proofs. We denote propositions by *form*. This constitutes a semantic subclass of the syntactic class *term*.

- Set is is the universe of *program types* or *specifications*. The specifications themselves are typing the programs. We denote specifications by *specif*. This constitutes a semantic subclass of the syntactic class *term*.

- Type is the type of Set and Prop

More on sorts can be found in Section 4.1.1.

COQ terms are typed. COQ types are recognized by the same syntactic class as *term*. We denote by *type* the semantic subclass of types inside the syntactic class *term*.

term	::=	`forall` *binderlist* `,` *term*	(1.2.8)
	|	`fun` *binderlist* `=>` *term*	(1.2.7)
	|	`fix` *fix_bodies*	(1.2.14)
	|	`cofix` *cofix_bodies*	(1.2.14)
	|	`let` *ident_with_params* `:=` *term* `in` *term*	(1.2.12)
	|	`let fix` *fix_body* `in` *term*	(1.2.14)
	|	`let cofix` *cofix_body* `in` *term*	(1.2.14)
	|	`let (` *[name , ... , name]* `)` *[dep_ret_type]* `:=` *term* `in` *term*	(1.2.13, 2.2.1)
	|	`if` *term [dep_ret_type]* `then` *term* `else` *term*	(1.2.13, 2.2.1)
	|	*term* `:` *term*	(1.2.10)
	|	*term* `->` *term*	(1.2.8)
	|	*term arg ... arg*	(1.2.9)
	|	`@` *qualid [term ... term]*	(2.7.11)
	|	*term* `%` *ident*	(12.2.2)
	|	`match` *match_item , ... , match_item [return_type]* `with` *[[|] equation | ... | equation]* `end`	(1.2.13)
	|	*qualid*	(1.2.3)
	|	*sort*	(1.2.5)
	|	*num*	(1.2.4)
	|	_	(1.2.11)
arg	::=	*term*	
	|	`(` *ident* `:=` *term* `)`	(2.7.11)
binderlist	::=	*name ... name [: term]*	1.2.6
	|	*binder binderlet ... binderlet*	
binder	::=	*name*	1.2.6
	|	`(` *name ... name* `:` *term* `)`	
binderlet	::=	*binder*	1.2.6
	|	`(` *name [: term]* `:=` *term* `)`	
name	::=	*ident*	
	|	_	
qualid	::=	*ident*	
	|	*qualid access_ident*	
sort	::=	`Prop` | `Set` | `Type`	

Figure 1.1: Syntax of terms

1.2.6 Binders

Various constructions such as `fun`, `forall`, `fix` and `cofix` *bind* variables. A binding is represented by an identifier. If the binding variable is not used in the expression, the identifier can be replaced by the

ident_with_params ::= *ident* [*binderlet* ... *binderlet*] [: *term*]

fix_bodies ::= *fix_body*
| *fix_body* with *fix_body* with ... with *fix_body* for *ident*

cofix_bodies ::= *cofix_body*
| *cofix_body* with *cofix_body* with ... with *cofix_body* for *ident*

fix_body ::= *ident binderlet* ... *binderlet* [*annotation*] [: *term*] := *term*

cofix_body ::= *ident_with_params* := *term*

annotation ::= { struct *ident* }

match_item ::= *term* [as *name*] [in *term*]

dep_ret_type ::= [as *name*] *return_type*

return_type ::= return *term*

equation ::= *mult_pattern* | ... | *mult_pattern* => *term*

mult_pattern ::= *pattern* , ... , *pattern*

pattern ::= *qualid pattern* ... *pattern*
| *pattern* as *ident*
| *pattern* % *ident*
| *qualid*
| _
| *num*
| (*or_pattern* , ... , *or_pattern*)

or_pattern ::= *pattern* | ... | *pattern*

Figure 1.2: Syntax of terms (continued)

symbol _. When the type of a bound variable cannot be synthesized by the system, it can be specified with the notation (*ident* : *type*). There is also a notation for a sequence of binding variables sharing the same type: ($ident_1 \ldots ident_n$: *type*).

Some constructions allow the binding of a variable to value. This is called a "let-binder". The entry *binderlet* of the grammar accepts either a binder as defined above or a let-binder. The notation in the latter case is (*ident* := *term*). In a let-binder, only one variable can be introduced at the same time. It is also possible to give the type of the variable as follows: (*ident* : *term* := *term*).

Lists of *binderlet* are allowed. In the case of fun and forall, the first binder of the list cannot be a let-binder, but parentheses can be omitted in the case of a single sequence of bindings sharing the same type (e.g.: fun (x y z : A) => t can be shortened in fun x y z : A => t).

1.2.7 Abstractions

The expression "`fun` *ident* `:` *type* `=>` *term*" defines the *abstraction* of the variable *ident*, of type *type*, over the term *term*. It denotes a function of the variable *ident* that evaluates to the expression *term* (e.g. `fun x:`A `=> x` denotes the identity function on type A). The keyword `fun` can be followed by several binders as given in Section 1.2.6. Functions over several variables are equivalent to an iteration of one-variable functions. For instance the expression "`fun` $ident_1 \ldots ident_n$ `:` *type* `=>` *term*" denotes the same function as "`fun` $ident_1$ `:` *type* `=>` ... `fun` $ident_n$ `:` *type* `=>` *term*". If a let-binder occurs in the list of binders, it is expanded to a local definition (see Section 1.2.12).

1.2.8 Products

The expression "`forall` *ident* `:` *type*`,` *term*" denotes the *product* of the variable *ident* of type *type*, over the term *term*. As for abstractions, `forall` is followed by a binder list, and products over several variables are equivalent to an iteration of one-variable products. Note that *term* is intended to be a type.

If the variable *ident* occurs in *term*, the product is called *dependent product*. The intention behind a dependent product `forall` x `:` A`,` B is twofold. It denotes either the universal quantification of the variable x of type A in the proposition B or the functional dependent product from A to B (a construction usually written $\Pi_{x:A}.B$ in set theory).

Non dependent product types have a special notation: "A -> B" stands for "`forall _:`A`,` B". The non dependent product is used both to denote the propositional implication and function types.

1.2.9 Applications

The expression $term_0$ $term_1$ denotes the application of $term_0$ to $term_1$.

The expression $term_0$ $term_1$ ... $term_n$ denotes the application of the term $term_0$ to the arguments $term_1$ ... then $term_n$. It is equivalent to `(` ... `(` $term_0$ $term_1$ `)` ... `)` $term_n$: associativity is to the left.

The notation `(` *ident* `:=` *term* `)` for arguments is used for making explicit the value of implicit arguments (see Section 2.7.11).

1.2.10 Type cast

The expression "*term* `:` *type*" is a type cast expression. It enforces the type of *term* to be *type*.

1.2.11 Inferable subterms

Expressions often contain redundant pieces of information. Subterms that can be automatically inferred by COQ can be replaced by the symbol "_" and COQ will guess the missing piece of information.

1.2.12 Local definitions (let-in)

`let` *ident* `:=` $term_1$ `in` $term_2$ denotes the local binding of $term_1$ to the variable *ident* in $term_2$. There is a syntactic sugar for local definition of functions: `let` *ident* $binder_1 \ldots binder_n$ `:=` $term_1$ `in` $term_2$ stands for `let` *ident* `:=` `fun` $binder_1 \ldots binder_n$ `=>` $term_2$ `in` $term_2$.

1.2.13 Definition by case analysis

Objects of inductive types can be destructurated by a case-analysis construction called *pattern-matching* expression. A pattern-matching expression is used to analyze the structure of an inductive objects and to apply specific treatments accordingly.

This paragraph describes the basic form of pattern-matching. See Section 2.2.1 and Chapter 16 for the description of the general form. The basic form of pattern-matching is characterized by a single *match_item* expression, a *mult_pattern* restricted to a single *pattern* and *pattern* restricted to the form *qualid ident* ... *ident*.

The expression `match` *term*$_0$ *return_type* `with` *pattern*$_1$ `=>` *term*$_1$ | ... | *pattern*$_n$ `=>` *term*$_n$ `end`, denotes a *pattern-matching* over the term *term*$_0$ (expected to be of an inductive type I). The terms *term*$_1$...*term*$_n$ are the *branches* of the pattern-matching expression. Each of *pattern*$_i$ has a form *qualid ident* ... *ident* where *qualid* must denote a constructor. There should be exactly one branch for every constructor of I.

The *return_type* expresses the type returned by the whole `match` expression. There are several cases. In the *non dependent* case, all branches have the same type, and the *return_type* is the common type of branches. In this case, *return_type* can usually be omitted as it can be inferred from the type of the branches[2].

In the *dependent* case, there are three subcases. In the first subcase, the type in each branch may depend on the exact value being matched in the branch. In this case, the whole pattern-matching itself depends on the term being matched. This dependency of the term being matched in the return type is expressed with an "`as` *ident*" clause where *ident* is dependent in the return type. For instance, in the following example:

```
Coq < Inductive bool : Type := true : bool | false : bool.

Coq < Inductive eq (A:Type) (x:A) : A -> Prop := refl_equal : eq A x x.

Coq < Inductive or (A:Prop) (B:Prop) : Prop :=
Coq < | or_introl : A -> or A B
Coq < | or_intror : B -> or A B.

Coq < Definition bool_case (b:bool) : or (eq bool b true) (eq bool b false)
Coq < := match b as x return or (eq bool x true) (eq bool x false) with
Coq <    | true  => or_introl (eq bool true true) (eq bool true false)
Coq <                 (refl_equal bool true)
Coq <    | false => or_intror (eq bool false true) (eq bool false false)
Coq <                 (refl_equal bool false)
Coq <    end.
```

the branches have respective types `or (eq bool true true) (eq bool true false)` and `or (eq bool false true) (eq bool false false)` while the whole pattern-matching expression has type `or (eq bool b true) (eq bool b false)`, the identifier `x` being used to represent the dependency. Remark that when the term being matched is a variable, the `as` clause can be omitted and the term being matched can serve itself as binding name in the return type. For instance, the following alternative definition is accepted and has the same meaning as the previous one.

```
Coq < Definition bool_case (b:bool) : or (eq bool b true) (eq bool b false)
Coq < := match b return or (eq bool b true) (eq bool b false) with
Coq <    | true  => or_introl (eq bool true true) (eq bool true false)
Coq <                 (refl_equal bool true)
Coq <    | false => or_intror (eq bool false true) (eq bool false false)
```

[2]Except if the inductive type is empty in which case there is no equation to help to infer the return type.

```
Coq <                  (refl_equal bool false)
Coq <    end.
```

The second subcase is only relevant for annotated inductive types such as the equality predicate (see Section 3.1.2), the order predicate on natural numbers or the type of lists of a given length (see Section 16.3). In this configuration, the type of each branch can depend on the type dependencies specific to the branch and the whole pattern-matching expression has a type determined by the specific dependencies in the type of the term being matched. This dependency of the return type in the annotations of the inductive type is expressed using a "`in I _ ... _` $ident_1$ `...` $ident_n$" clause, where

- I is the inductive type of the term being matched;
- the names $ident_i$'s correspond to the arguments of the inductive type that carry the annotations: the return type is dependent on them;
- the _'s denote the family parameters of the inductive type: the return type is not dependent on them.

For instance, in the following example:

```
Coq < Definition sym_equal (A:Type) (x y:A) (H:eq A x y) : eq A y x :=
Coq <   match H in eq _ _ z return eq A z x with
Coq <   | refl_equal => refl_equal A x
Coq <   end.
```

the type of the branch has type `eq A x x` because the third argument of `eq` is `x` in the type of the pattern `refl_equal`. On the contrary, the type of the whole pattern-matching expression has type `eq A y x` because the third argument of `eq` is `y` in the type of `H`. This dependency of the case analysis in the third argument of `eq` is expressed by the identifier `z` in the return type.

Finally, the third subcase is a combination of the first and second subcase. In particular, it only applies to pattern-matching on terms in a type with annotations. For this third subcase, both the clauses `as` and `in` are available.

There are specific notations for case analysis on types with one or two constructors: "`if ... then ... else ...`" and "`let (... , ... , ...) := ... in ...`" (see Sections 2.2.2 and 2.2.3).

1.2.14 Recursive functions

The expression "`fix` $ident_1$ $binder_1$ `:` $type_1$ `:=` $term_1$ `with ... with` $ident_n$ $binder_n$ `:` $type_n$ `:=` $term_n$ `for` $ident_i$" denotes the i^{th}component of a block of functions defined by mutual well-founded recursion. It is the local counterpart of the `Fixpoint` command. See Section 1.3.4 for more details. When $n = 1$, the "`for` $ident_i$" clause is omitted.

The expression "`cofix` $ident_1$ $binder_1$ `:` $type_1$ `with ... with` $ident_n$ $binder_n$ `:` $type_n$ `for` $ident_i$" denotes the i^{th}component of a block of terms defined by a mutual guarded co-recursion. It is the local counterpart of the `CoFixpoint` command. See Section 1.3.4 for more details. When $n = 1$, the "`for` $ident_i$" clause is omitted.

The association of a single fixpoint and a local definition have a special syntax: "`let fix` f `... := ... in ...`" stands for "`let` f `:= fix` f `... := ... in ...`". The same applies for co-fixpoints.

sentence ::= *declaration*
| *definition*
| *inductive*
| *fixpoint*
| *statement [proof]*

declaration ::= *declaration_keyword assums* .

declaration_keyword ::= `Axiom` | `Conjecture`
| `Parameter` | `Parameters`
| `Variable` | `Variables`
| `Hypothesis` | `Hypotheses`

assums ::= *ident* ... *ident* : *term*
| *binder* ... *binder*

definition ::= `Definition` *ident_with_params* := *term* .
| `Let` *ident_with_params* := *term* .

inductive ::= `Inductive` *ind_body* `with` ... `with` *ind_body* .
| `CoInductive` *ind_body* `with` ... `with` *ind_body* .

ind_body ::= *ident [binderlet* ... *binderlet]* : *term* :=
[[|] ident_with_params | ... | *ident_with_params]*

fixpoint ::= `Fixpoint` *fix_body* `with` ... `with` *fix_body* .
| `CoFixpoint` *cofix_body* `with` ... `with` *cofix_body* .

statement ::= *statement_keyword ident [binderlet* ... *binderlet]* : *term* .

statement_keyword ::= `Theorem` | `Lemma` | `Definition`

proof ::= `Proof` . ... `Qed` .
| `Proof` . ... `Defined` .
| `Proof` . ... `Admitted` .

Figure 1.3: Syntax of sentences

1.3 The Vernacular

Figure 1.3 describes *The Vernacular* which is the language of commands of GALLINA. A sentence of the vernacular language, like in many natural languages, begins with a capital letter and ends with a dot.

The different kinds of command are described hereafter. They all suppose that the terms occurring in the sentences are well-typed.

1.3.1 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 if and only if this *term* is a correct type in the current context of the declaration and *ident* was not previously defined in the same module. This *term* is considered to be the type, or specification, of the *ident*.

`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 messages:

1. *ident* `already exists`

Variants:

1. `Parameter` *ident* : *term* .
   Is equivalent to `Axiom` *ident* : *term*
2. `Parameter` $ident_1 \ldots ident_n$: *term* .
   Adds n parameters with specification *term*
3. `Parameter` ($ident_{1,1} \ldots ident_{1,k_1}$: $term_1$) ... ($ident_{n,1} \ldots ident_{n,k_n}$: $term_n$) .
   Adds n blocks of parameters with different specifications.
4. `Conjecture` *ident* : *term* .
   Is equivalent to `Axiom` *ident* : *term*.

Remark: It is possible to replace `Parameter` by `Parameters`.

`Variable` *ident* : *term*.

This command links *term* to the name *ident* in the context of the current section (see Section 2.4 for a description of the section mechanism). When the current section is closed, name *ident* will be unknown and every object using this variable will be explicitly parametrized (the variable is *discharged*). Using the `Variable` command out of any section is equivalent to `Axiom`.

Error messages:

1. *ident* `already exists`

Variants:

1. `Variable` $ident_1 \ldots ident_n$: *term* .
   Links *term* to names $ident_1 \ldots ident_n$.
2. `Variable` ($ident_{1,1} \ldots ident_{1,k_1}$: $term_1$) ... ($ident_{n,1} \ldots ident_{n,k_n}$: $term_n$) .
   Adds n blocks of variables with different specifications.

3. `Hypothesis` *ident* `:` *term* `.`
   `Hypothesis` is a synonymous of `Variable`

Remark: It is possible to replace `Variable` by `Variables` and `Hypothesis` by `Hypotheses`.

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).

1.3.2 Definitions

Definitions differ from declarations in allowing 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 4.3). A defined object is accepted by the system if and only if 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 environments is given in Section 4.2.

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

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

Error messages:

1. *ident* `already exists`

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 $term_2$.

2. `Definition` *ident* $binder_1 \ldots binder_n$ `:` $term_1$ `:=` $term_2$ `.`
   This is equivalent to
   `Definition` *ident* `:` `forall` $binder_1 \ldots binder_n$`,` $term_1$ `:=` `fun` $binder_1 \ldots binder_n$ `=>` $term_2$ `.`

3. `Example` *ident* `:=` *term* `.`
   `Example` *ident* `:` $term_1$ `:=` $term_2$ `.`
   `Example` *ident* $binder_1 \ldots binder_n$ `:` $term_1$ `:=` $term_2$ `.`
   These are synonyms of the `Definition` forms.

Error messages:

1. `Error: The term "`*term*`" has type "`*type*`" while it is expected to have type "`*type*`"`

See also: Sections 6.9.1, 6.9.2, 8.5.5

`Let` *ident* `:=` *term* `.`

This command binds the value *term* to the name *ident* in the environment of the current section. The name *ident* disappears when the current section is eventually closed, and, all persistent objects (such as theorems) defined within the section and depending on *ident* are prefixed by the local definition `let` *ident* `:=` *term* `in`.

Error messages:

1. *ident* `already exists`

Variants:

1. `Let` *ident* `:` $term_1$ `:=` $term_2$ `.`

See also: Sections 2.4 (section mechanism), 6.9.1, 6.9.2 (opaque/transparent constants), 8.5.5

1.3.3 Inductive definitions

We gradually explain simple inductive types, simple annotated inductive types, simple parametric inductive types, mutually inductive types. We explain also co-inductive types.

Simple inductive types

The definition of a simple inductive type has the following form:

```
Inductive ident : sort :=
    ident_1 : type_1
  | ...
  | ident_n : type_n
```

The name *ident* is the name of the inductively defined type and *sort* is the universes where it lives. The names $ident_1$, ..., $ident_n$ are the names of its constructors and $type_1$, ..., $type_n$ their respective types. The types of the constructors have to satisfy a *positivity condition* (see Section 4.5.3) for *ident*. This condition ensures the soundness 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. Accordingly to the universe where the inductive type lives (*e.g.* its type *sort*), 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`. The type of the destructors expresses structural induction/recursion principles over objects of *ident*. We give below two examples of the use of the `Inductive` definitions.

The set of natural numbers is defined as:

```
Coq < Inductive nat : Set :=
Coq <   | O : nat
Coq <   | S : nat -> nat.
nat is defined
nat_rect is defined
nat_ind is defined
nat_rec is defined
```

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 of them: nat_ind, nat_rec and nat_rect. The type of nat_ind is:

```
Coq < Check nat_ind.
nat_ind
     : forall P : nat -> Prop,
       P O -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

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 (forall n:nat, P n) by induction on 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. The constant *ident*_ind is always provided, whereas *ident*_rec and *ident*_rect can be impossible to derive (for example, when *ident* is a proposition).

Variants:

1. Coq < Inductive nat : Set := O | S (_:nat).
   In the case where inductive types have no annotations (next section gives an example of such annotations), a constructor can be defined by only giving the type of its arguments.

Simple annotated inductive types

In an annotated inductive types, the universe where the inductive type is defined is no longer a simple sort, but what is called an arity, which is a type whose conclusion is a sort.

As an example of annotated inductive types, let us define the $even$ predicate:

```
Coq < Inductive even : nat -> Prop :=
Coq <   | even_0 : even O
Coq <   | even_SS : forall n:nat, even n -> even (S (S n)).
even is defined
even_ind is defined
```

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.
even_ind
     : forall P : nat -> Prop,
       P O ->
       (forall n : nat, even n -> P n -> P (S (S n))) ->
       forall n : nat, even n -> P n
```

From a mathematical point of view it asserts that the natural numbers satisfying the predicate even are exactly in the smallest set of 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 messages:

1. `Non strictly positive occurrence of` *ident* `in` *type*
2. `The conclusion of` *type* `is not valid; it must be built from` *ident*

Parametrized inductive types

In the previous example, each constructor introduces a different instance of the predicate `even`. In some cases, all the constructors introduces the same generic instance of the inductive definition, in which case, instead of an annotation, we use a context of parameters which are binders shared by all the constructors of the definition.

The general scheme is:

`Inductive` *ident binder*$_1$ ... *binder*$_k$: *term* := *ident*$_1$: *term*$_1$ | ... | *ident*$_n$: *term*$_n$.

Parameters differ from inductive type annotations in the fact that the conclusion of each type of constructor *term*$_i$ invoke the inductive type with the same values of parameters as its specification.

A typical example is the definition of polymorphic lists:

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

Note that in the type of `nil` and `cons`, we write `(list A)` and not just `list`.
The constants `nil` and `cons` will have respectively types:

```
Coq < Check nil.
nil
     : forall A : Set, list A

Coq < Check cons.
cons
     : forall A : Set, A -> list A -> list A
```

Types of destructors are also quantified with `(A:Set)`.

Variants:

1. `Coq < Inductive list (A:Set) : Set := nil | cons (_:A) (_:list A).`
   This is an alternative definition of lists where we specify the arguments of the constructors rather than their full type.

Error messages:

1. `The` *num*`th argument of` *ident* `must be` *ident'* `in` *type*

New from Coq V8.1 The condition on parameters for inductive definitions has been relaxed since Coq V8.1. It is now possible in the type of a constructor, to invoke recursively the inductive definition on an argument which is not the parameter itself.

One can define :

```
Coq < Inductive list2 (A:Set) : Set :=
Coq <   | nil2 : list2 A
Coq <   | cons2 : A -> list2 (A*A) -> list2 A.
list2 is defined
list2_rect is defined
list2_ind is defined
list2_rec is defined
```

that can also be written by specifying only the type of the arguments:

```
Coq < Inductive list2 (A:Set) : Set := nil2 | cons2 (_:A) (_:list2 (A*A)).
```

But the following definition will give an error:

```
Coq < Inductive listw (A:Set) : Set :=
Coq <   | nilw : listw (A*A)
Coq <   | consw : A -> listw (A*A) -> listw (A*A).
Error: Last occurrence of "listw" must have "A" as 1st argument in
 "listw (A * A)%type".
```

Because the conclusion of the type of constructors should be `listw A` in both cases.

A parametrized inductive definition can be defined using annotations instead of parameters but it will sometimes give a different (bigger) sort for the inductive definition and will produce a less convenient rule for case elimination.

See also: Sections 4.5 and 8.7.

Mutually defined inductive types

The definition of a block of mutually inductive types has the form:

```
Inductive ident_1 : type_1 :=
    ident_1^1 : type_1^1
  | ...
  | ident_{n_1}^1 : type_{n_1}^1
with
  ...
with ident_m : type_m :=
    ident_1^m : type_1^m
  | ...
  | ident_{n_m}^m : type_{n_m}^m .
```

It has the same semantics as the above `Inductive` definition for each $ident_1, \ldots, ident_m$. All names $ident_1, \ldots, ident_m$ and $ident_1^1, \ldots, ident_{n_m}^m$ are simultaneously added to the environment. Then well-typing of constructors can be checked. Each one of the $ident_1, \ldots, ident_m$ can be used on its own.

It is also possible to parametrize these inductive definitions. However, parameters correspond to a local context in which the whole set of inductive declarations is done. For this reason, the parameters must be strictly the same for each inductive types The extended syntax is:

```
Inductive ident_1 params : type_1 :=
    ident_1^1 : type_1^1
  | ...
  | ident_{n_1}^1 : type_{n_1}^1
with
  ...
with ident_m params : type_m :=
    ident_1^m : type_1^m
  | ...
  | ident_{n_m}^m : type_{n_m}^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 < Inductive tree : Set :=
Coq <     node : A -> forest -> tree
Coq < with forest : Set :=
Coq <   | leaf : B -> forest
Coq <   | cons : tree -> forest -> forest.
```

This declaration generates automatically six induction principles. They are respectively called `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.
tree_rec
     : forall P : tree -> Set,
       (forall (a : A) (f : forest), P (node a f)) -> forall t : tree, P t

Coq < Check forest_rec.
forest_rec
     : forall P : forest -> Set,
       (forall b : B, P (leaf b)) ->
       (forall (t : tree) (f0 : forest), P f0 -> P (cons t f0)) ->
       forall f1 : forest, P f1
```

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

```
Coq < Inductive tree (A B:Set) : Set :=
Coq <     node : A -> forest A B -> tree A B
Coq < with forest (A B:Set) : Set :=
Coq <   | 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 occurring free in the declaration are added as parameters to the inductive definition.

See also: Section 2.4

Co-inductive types

The objects of an inductive type are well-founded with respect to the constructors of the type. In other words, such objects contain only a *finite* number of constructors. Co-inductive types arise from relaxing this condition, and admitting types whose objects contain an infinity of constructors. Infinite objects are introduced by a non-ending (but effective) process of construction, defined in terms of the constructors of the type.

An example of a co-inductive type is the type of infinite sequences of natural numbers, usually called streams. It can be introduced in Coq using the `CoInductive` command:

```
Coq < CoInductive Stream : Set :=
Coq <     Seq : nat -> Stream -> Stream.
Stream is defined
```

The syntax of this command is the same as the command `Inductive` (see Section 1.3.3). Notice that no principle of induction is derived from the definition of a co-inductive type, since such principles only make sense for inductive ones. For co-inductive ones, the only elimination principle is case analysis. For example, the usual destructors on streams `hd:Stream->nat` and `tl:Str->Str` can be defined as follows:

```
Coq < Definition hd (x:Stream) := let (a,s) := x in a.
hd is defined
Coq < Definition tl (x:Stream) := let (a,s) := x in s.
tl is defined
```

Definition of co-inductive predicates and blocks of mutually co-inductive definitions are also allowed. An example of a co-inductive predicate is the extensional equality on streams:

```
Coq < CoInductive EqSt : Stream -> Stream -> Prop :=
Coq <     eqst :
Coq <       forall s1 s2:Stream,
Coq <         hd s1 = hd s2 -> EqSt (tl s1) (tl s2) -> EqSt s1 s2.
EqSt is defined
```

In order to prove the extensionally equality of two streams s_1 and s_2 we have to construct an infinite proof of equality, that is, an infinite object of type $(\texttt{EqSt}\ s_1\ s_2)$. We will see how to introduce infinite objects in Section 1.3.4.

1.3.4 Definition of recursive functions

Definition of functions by recursion over inductive objects

This section describes the primitive form of definition by recursion over inductive objects. See Section 2.3 for more advanced constructions. The command:

`Fixpoint` *ident params* `{struct` *ident*$_0$ `}` `:` type$_0$ `:=` *term*$_0$

allows to define functions by pattern-matching over inductive objects using a fixed point construction. The meaning of this declaration is to define *ident* a recursive function with arguments specified by the binders in *params* such that *ident* applied to arguments corresponding to these binders has type *type*$_0$, and is equivalent to the expression *term*$_0$. The type of the *ident* is consequently `forall` *params* `,` *type*$_0$ and the value is equivalent to `fun` *params* `=>` *term*$_0$.

To be accepted, a `Fixpoint` definition has to satisfy some syntactical constraints on a special argument called the decreasing argument. They are needed to ensure that the `Fixpoint` definition always terminates. The point of the `{struct` *ident* `}` annotation is to let the user tell the system which argument decreases along the recursive calls. For instance, one can define the addition function as :

```
Coq < Fixpoint add (n m:nat) {struct n} : nat :=
Coq <   match n with
Coq <   | O => m
Coq <   | S p => S (add p m)
Coq <   end.
add is recursively defined (decreasing on 1st argument)
```

The `{struct` *ident* `}` annotation may be left implicit, in this case the system try successively arguments from left to right until it finds one that satisfies the decreasing condition. Note that some fixpoints may have several arguments that fit as decreasing arguments, and this choice influences the reduction of the fixpoint. Hence an explicit annotation must be used if the leftmost decreasing argument is not the desired one. Writing explicit annotations can also speed up type-checking of large mutual fixpoints.

The `match` 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 `(S (add p m))`.

The `match` operator is formally described in detail in Section 4.5.4. The system recognizes that in the inductive call `(add p m)` the first argument actually decreases because it is a *pattern variable* coming from `match n with`.

Example: The following definition is not correct and generates an error message:

```
Coq < Fixpoint wrongplus (n m:nat) {struct n} : nat :=
Coq <   match m with
Coq <   | O => n
Coq <   | S p => S (wrongplus n p)
Coq <   end.
Coq < Coq < Error:
Recursive definition of wrongplus is ill-formed.
In environment
wrongplus : nat -> nat -> nat
n : nat
m : nat
p : nat
Recursive call to wrongplus has principal argument equal to
"n"
instead of a subterm of n.
```

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) {struct m} : nat :=
Coq <   match m with
Coq <   | O => n
Coq <   | S p => S (plus n p)
Coq <   end.
```

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

```
Coq < Fixpoint nat_match
Coq <   (C:Set) (f0:C) (fS:nat -> C -> C) (n:nat) {struct n} : C :=
Coq <   match n with
Coq <   | O => f0
Coq <   | S p => fS p (nat_match C f0 fS p)
Coq <   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 <   match n with
Coq <   | O => O
Coq <   | S p => match p with
Coq <             | O => S O
Coq <             | S q => mod2 q
Coq <             end
Coq <   end.
```

In order to keep the strong normalization property, the fixed point reduction will only be performed when the argument in position of the decreasing argument (which 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.

Variants:

1. `Fixpoint` $ident_1$ $params_1$ `:`$type_1$ `:=` $term_1$
   `with ...`
   `with` $ident_m$ $params_m$ `:`$type_m$ `:=` $term_m$
   Allows to define simultaneously $ident_1, \ldots, ident_m$.

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

```
Coq < Fixpoint tree_size (t:tree) : nat :=
Coq <   match t with
Coq <   | node a f => S (forest_size f)
Coq <   end
Coq <  with forest_size (f:forest) : nat :=
Coq <   match f with
Coq <   | leaf b => 1
Coq <   | cons t f' => (tree_size t + forest_size f')
Coq <   end.
```

A generic command `Scheme` is useful to build automatically various mutual induction principles. It is described in Section 8.14.

Definition of recursive objects in co-inductive types

The command:

CoFixpoint *ident* : *type*$_0$:= *term*$_0$

introduces a method for constructing an infinite object of a coinductive type. For example, the stream containing all natural numbers can be introduced applying the following method to the number O (see Section 1.3.3 for the definition of Stream, hd and tl):

```
Coq < CoFixpoint from (n:nat) : Stream := Seq n (from (S n)).
from is corecursively defined
```

Oppositely to recursive ones, there is no decreasing argument in a co-recursive definition. To be admissible, a method of construction must provide at least one extra constructor of the infinite object for each iteration. A syntactical guard condition is imposed on co-recursive definitions in order to ensure this: each recursive call in the definition must be protected by at least one constructor, and only by constructors. That is the case in the former definition, where the single recursive call of from is guarded by an application of Seq. On the contrary, the following recursive function does not satisfy the guard condition:

```
Coq < CoFixpoint filter (p:nat -> bool) (s:Stream) : Stream :=
Coq <   if p (hd s) then Seq (hd s) (filter p (tl s)) else filter p (tl s).
Coq < Coq < Error:
Recursive definition of filter is ill-formed.
In environment
filter : (nat -> bool) -> Stream -> Stream
p : nat -> bool
s : Stream
Unguarded recursive call in "filter p (tl s)".
```

The elimination of co-recursive definition is done lazily, i.e. the definition is expanded only when it occurs at the head of an application which is the argument of a case analysis expression. In any other context, it is considered as a canonical expression which is completely evaluated. We can test this using the command Eval, which computes the normal forms of a term:

```
Coq < Eval compute in (from 0).
     = (cofix from (n : nat) : Stream := Seq n (from (S n))) 0
     : Stream
Coq < Eval compute in (hd (from 0)).
     = 0
     : nat
Coq < Eval compute in (tl (from 0)).
     = (cofix from (n : nat) : Stream := Seq n (from (S n))) 1
     : Stream
```

Variants:

1. CoFixpoint *ident*$_1$ *params* :*type*$_1$:= *term*$_1$
   As for most constructions, arguments of co-fixpoints expressions can be introduced before the := sign.
2. CoFixpoint *ident*$_1$:*type*$_1$:= *term*$_1$
   with
   ...
   with *ident*$_m$: *type*$_m$:= *term*$_m$
   As in the Fixpoint command (see Section 1.3.4), it is possible to introduce a block of mutually dependent methods.

1.3.5 Statement and proofs

A statement claims a goal of which the proof is then interactively done using tactics. More on the proof editing mode, statements and proofs can be found in Chapter 7.

`Theorem` *ident* `:` *type* `.`

This command binds *type* to the name *ident* in the environment, provided that a proof of *type* is next given.

After a statement, Coq needs a proof.

Variants:

1. `Lemma` *ident* `:` *type* `.`
   `Remark` *ident* `:` *type* `.`
   `Fact` *ident* `:` *type* `.`
   `Corollary` *ident* `:` *type* `.`
   `Proposition` *ident* `:` *type* `.`
   All these commands are synonymous of `Theorem`

2. `Theorem` *ident* `:` *type* `with ... with` *ident* `:` *type* `.`

   This command is useful for theorems that are proved by simultaneous induction over a mutually inductive assumption, or that state mutually dependent statements in some mutual coinductive type. It is equivalent to `Fixpoint` (see Section 1.3.4) or `CoFixpoint` (see Section 1.3.4) but using tactics to build the proof of the statements (or the body of the specification, depending on the point of view). The inductive or coinductive types on which the induction or coinduction has to be done is assumed to be non ambiguous and is guessed by the system.

   Like in a `Fixpoint` or `CoFixpoint` definition, the induction hypotheses have to be used on *structurally smaller* arguments (for a `Fixpoint`) or be *guarded by a constructor* (for a `CoFixpoint`). The verification that recursive proof arguments are correct is done only at the time of registering the lemma in the environment. To know if the use of induction hypotheses is correct at some time of the interactive development of a proof, use the command `Guarded` (see Section 7.3.2).

   The command can be used also with `Lemma`, `Remark`, etc. instead of `Theorem`.

3. `Definition` *ident* `:` *type* `.`
   Allow to define a term of type *type* using the proof editing mode. It behaves as `Theorem` but is intended for the interactive definition of expression which computational behavior will be used by further commands. **See also:** 6.9.2 and 8.5.5.

`Proof .` **...** `Qed .`

A proof starts by the keyword `Proof`. Then Coq enters the proof editing mode until the proof is completed. The proof editing mode essentially contains tactics that are described in chapter 8. Besides tactics, there are commands to manage the proof editing mode. They are described in Chapter 7. When the proof is completed it should be validated and put in the environment using the keyword `Qed`.

Error message:

1. *ident* `already exists`

Remarks:

1. Several statements can be simultaneously opened.
2. Not only other statements but any vernacular command can be given within the proof editing mode. In this case, the command is understood as if it would have been given before the statements still to be proved.
3. `Proof` is recommended but can currently be omitted. On the opposite, `Qed` (or `Defined`, see below) is mandatory to validate a proof.
4. Proofs ended by `Qed` are declared opaque (see 6.9.1) and cannot be unfolded by conversion tactics (see 8.5). To be able to unfold a proof, you should end the proof by `Defined` (see below).

Variants:

1. `Proof .` ...`Defined .`
   Same as `Proof .` ...`Qed .` but the proof is then declared transparent (see 6.9.2), which means it can be unfolded in conversion tactics (see 8.5).
2. `Proof .` ...`Save.`
   Same as `Proof .` ...`Qed .`
3. `Goal` *type*...`Save` *ident*
   Same as `Lemma` *ident* `:` *type*...`Save.` This is intended to be used in the interactive mode. Conversely to named lemmas, anonymous goals cannot be nested.
4. `Proof.` ...`Admitted.`
   Turns the current conjecture into an axiom and exits editing of current proof.

Chapter 2

Extensions of GALLINA

GALLINA is the kernel language of COQ. We describe here extensions of the Gallina's syntax.

2.1 Record types

The `Record` construction is a macro allowing the definition of records as is done in many programming languages. Its syntax is described on Figure 2.1. In fact, the `Record` macro is more general than the usual record types, since it allows also for "manifest" expressions. In this sense, the `Record` construction allows to define "signatures".

sentence	++=	*record*
record	::=	*inductivity_token* *ident* [*binderlet* ... *binderlet*] [: *sort*] := [*ident*] { [*field* ; ... ; *field*] } .
inductivity_token	::=	`Record` | `Structure` | `Inductive` | `CoInductive`
field	::=	*name* : *type* [`where` *notation*]
	|	*name* [: *term*] := *term*

Figure 2.1: Syntax for the definition of `Record`

In the expression

`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. If *ident*$_0$ is omitted, the default name `Build_`*ident* is used. If *sort* is omitted, the default sort is "Type". The identifiers *ident*$_1$, .., *ident*$_n$ are the names of fields and *term*$_1$, .., *term*$_n$ their respective types. Remark that the type of *ident*$_i$ may depend on the previous *ident*$_j$ (for $j < i$). Thus the order of the fields is important. Finally, *params* are the parameters of the record.

More generally, a record may have explicitly defined (a.k.a. manifest) fields. For instance, `Record` *ident* [*params*] : *sort* := { *ident*$_1$: *type*$_1$; *ident*$_2$:= *term*$_2$; *ident*$_3$: *type*$_3$ } in which case the correctness of *type*$_3$ may rely on the instance *term*$_2$ of *ident*$_2$ and *term*$_2$ in turn may depend on *ident*$_1$.

Example: The set of rational numbers may be defined as:

```
Coq < Record Rat : Set := mkRat
Coq <   {sign : bool;
Coq <    top : nat;
Coq <    bottom : nat;
Coq <    Rat_bottom_cond : 0 <> bottom;
Coq <    Rat_irred_cond :
Coq <     forall x y z:nat, (x * y) = top /\ (x * z) = bottom -> x = 1}.
Rat is defined
Rat_rect is defined
Rat_ind is defined
Rat_rec is defined
sign is defined
top is defined
bottom is defined
Rat_bottom_cond is defined
Rat_irred_cond is defined
```

Remark here that the field `Rat_cond` depends on the field `bottom`.

Let us now see the work done by the `Record` macro. First the macro generates an inductive definition with just one constructor:

`Inductive` *ident params* `:` *sort* `:=`
ident$_0$ `(`*ident*$_1$ `:` *term*$_1$`)` `..` `(`*ident*$_n$ `:` *term*$_n$`).`

To build an object of type *ident*, one should provide the constructor *ident*$_0$ with n terms filling the fields of the record.

As an example, let us define the rational $1/2$:

```
Coq < Require Import Arith.
Coq < Theorem one_two_irred :
Coq <  forall x y z:nat, x * y = 1 /\ x * z = 2 -> x = 1.
...
Coq < Qed.
Coq < Definition half := mkRat true 1 2 (O_S 1) one_two_irred.
half is defined
Coq < Check half.
half
     : Rat
```

The macro generates also, when it is possible, the projection functions for destructuring an object of type *ident*. These projection functions have the same name that the corresponding fields. If a field is named "_" then no projection is built for it. In our example:

```
Coq < Eval compute in half.(top).
     = 1
     : nat
Coq < Eval compute in half.(bottom).
     = 2
     : nat
Coq < Eval compute in half.(Rat_bottom_cond).
     = O_S 1
     : 0 <> bottom half
```

Variants:

1. Records declared with the keyword `Record` (or `Structure` which is equivalent to the former) cannot be recursive. However, records can be declared using the keyword `CoInductive` (resp. `Inductive`) instead, making them coinductive (resp. inductive) types.

   As an example, here is how to define a type of streams over a type A as a type with a pair of destructors:

```
Coq < CoInductive stream (A:Type) :=
Coq <   {head : A;
Coq <    tail : stream A}.
stream is defined
head is defined
tail is defined
```

Warnings:

1. `Warning:` *ident*$_i$ `cannot be defined.`

   It can happen that the definition of a projection is impossible. This message is followed by an explanation of this impossibility. There may be three reasons:

   (a) The name *ident*$_i$ already exists in the environment (see Section 1.3.1).

   (b) The body of *ident*$_i$ uses an incorrect elimination for *ident* (see Sections 1.3.4 and 4.5.4).

   (c) The type of the projections *ident*$_i$ depends on previous projections which themselves couldn't be defined.

Error messages:

1. `A record cannot be recursive`

   The record name *ident* appears in the type of its fields.

2. During the definition of the one-constructor inductive definition, all the errors of inductive definitions, as described in Section 1.3.3, may also occur.

See also: Coercions and records in Section 17.9 of the chapter devoted to coercions.

Remark: `Structure` is a synonym of the keyword `Record`.

Remark: An experimental syntax for projections based on a dot notation is available. The command to activate it is

```
Set Printing Projections.
```

The corresponding grammar rules are given Figure 2.2. When *qualid* denotes a projection, the syntax *term*`.(`*qualid*`)` is equivalent to *qualid term*, the syntax *term*`.(`*qualid* *arg*$_1$ ... *arg*$_n$`)` to *qualid* *arg*$_1$ ... *arg*$_n$ *term*, and the syntax *term*`.(@`*qualid* *term*$_1$ ... *term*$_n$`)` to `@`*qualid* *term*$_1$ ... *term*$_n$ *term*. In each case, *term* is the object projected and the other arguments are the parameters of the inductive type.

To deactivate the printing of projections, use `Unset Printing Projections.`

term	++=	*term* . (*qualid*)
	|	*term* . (*qualid arg* ... *arg*)
	|	*term* . (@*qualid term* ... *term*)

Figure 2.2: Syntax of `Record` projections

2.2 Variants and extensions of `match`

2.2.1 Multiple and nested pattern-matching

The basic version of `match` allows pattern-matching on simple patterns. As an extension, multiple nested patterns or disjunction of patterns are allowed, as in ML-like languages.

The extension just acts as a macro that is expanded during parsing into a sequence of `match` on simple patterns. Especially, a construction defined using the extended `match` is generally printed under its expanded form (see `Set Printing Matching` in section 2.2.4).

See also: Chapter 16.

2.2.2 Pattern-matching on boolean values: the `if` expression

For inductive types with exactly two constructors and for pattern-matchings expressions which do not depend on the arguments of the constructors, it is possible to use a `if ... then ... else` notation. For instance, the definition

```
Coq < Definition not (b:bool) :=
Coq <   match b with
Coq <   | true => false
Coq <   | false => true
Coq <   end.
not is defined
```

can be alternatively written

```
Coq < Definition not (b:bool) := if b then false else true.
not is defined
```

More generally, for an inductive type with constructors C_1 and C_2, we have the following equivalence

if term [dep_ret_type] `then` *term*$_1$ `else` *term*$_2 \equiv$

```
match term [dep_ret_type] with
| C1 _ ... _ => term1
| C2 _ ... _ => term2
end
```

Here is an example.

```
Coq < Check (fun x (H:{x=0}+{x<>0}) =>
Coq <   match H with
Coq <   | left _ => true
Coq <   | right _ => false
Coq <   end).
fun (x : nat) (H : {x = 0} + {x <> 0}) => if H then true else false
     : forall x : nat, {x = 0} + {x <> 0} -> bool
```

Notice that the printing uses the `if` syntax because `sumbool` is declared as such (see Section 2.2.4).

2.2.3 Irrefutable patterns: the destructuring let variants

Pattern-matching on terms inhabiting inductive type having only one constructor can be alternatively written using `let ... in ...` constructions. There are two variants of them.

First destructuring let syntax

The expression `let (` *ident*$_1$,...,*ident*$_n$ `) :=` *term*$_0$ `in` *term*$_1$ performs case analysis on a *term*$_0$ which must be in an inductive type with one constructor having itself n arguments. Variables *ident*$_1$...*ident*$_n$ are bound to the n arguments of the constructor in expression *term*$_1$. For instance, the definition

```
Coq < Definition fst (A B:Set) (H:A * B) := match H with
Coq <                                         | pair x y => x
Coq <                                         end.
fst is defined
```

can be alternatively written

```
Coq < Definition fst (A B:Set) (p:A * B) := let (x, _) := p in x.
fst is defined
```

Notice that reduction is different from regular `let ... in ...` construction since it happens only if *term*$_0$ is in constructor form. Otherwise, the reduction is blocked.

The pretty-printing of a definition by matching on a irrefutable pattern can either be done using `match` or the `let` construction (see Section 2.2.4).

If *term* inhabits an inductive type with one constructor `C`, we have an equivalence between

`let (`*ident*$_1$`, ..., `*ident*$_n$`) [`*dep_ret_type*`] :=` *term* `in` *term'*

and

`match` *term* `[`*dep_ret_type*`] with C` *ident*$_1$ `...` *ident*$_n$ `=>` *term'* `end`

Second destructuring let syntax

Another destructuring `let` syntax is available for inductive types with one constructor by giving an arbitrary pattern instead of just a tuple for all the arguments. For example, the preceding example can be written:

```
Coq < Definition fst (A B:Set) (p:A * B) := let 'pair x _ := p in x.
fst is defined
```

This is useful to match deeper inside tuples and also to use notations for the pattern, as the syntax `let 'p := t in b` allows arbitrary patterns to do the deconstruction. For example:

```
Coq < Definition deep_tuple (A : Set) (x : (A * A) * (A * A)) : A * A * A * A :=
Coq <   let '((a,b), (c, d)) := x in (a,b,c,d).
deep_tuple is defined
Coq < Notation " x 'with' p " := (exist _ x p) (at level 20).
Coq < Definition proj1_sig' (A :Set) (P : A -> Prop) (t:{ x : A | P x }) : A :=
Coq <   let 'x with p := t in x.
proj1_sig' is defined
```

When printing definitions which are written using this construct it takes precedence over `let` printing directives for the datatype under consideration (see Section 2.2.4).

2.2.4 Controlling pretty-printing of `match` expressions

The following commands give some control over the pretty-printing of `match` expressions.

Printing nested patterns

The Calculus of Inductive Constructions knows pattern-matching only over simple patterns. It is however convenient to re-factorize nested pattern-matching into a single pattern-matching over a nested pattern. COQ's printer try to do such limited re-factorization.

```
Set Printing Matching.
```

This tells COQ to try to use nested patterns. This is the default behavior.

```
Unset Printing Matching.
```

This tells COQ to print only simple pattern-matching problems in the same way as the COQ kernel handles them.

```
Test Printing Matching.
```

This tells if the printing matching mode is on or off. The default is on.

Printing of wildcard pattern

Some variables in a pattern may not occur in the right-hand side of the pattern-matching clause. There are options to control the display of these variables.

```
Set Printing Wildcard.
```

The variables having no occurrences in the right-hand side of the pattern-matching clause are just printed using the wildcard symbol "_".

```
Unset Printing Wildcard.
```

The variables, even useless, are printed using their usual name. But some non dependent variables have no name. These ones are still printed using a "_".

```
Test Printing Wildcard.
```

This tells if the wildcard printing mode is on or off. The default is to print wildcard for useless variables.

Printing of the elimination predicate

In most of the cases, the type of the result of a matched term is mechanically synthesizable. Especially, if the result type does not depend of the matched term.

```
Set Printing Synth.
```

The result type is not printed when COQ knows that it can re-synthesize it.

```
Unset Printing Synth.
```

This forces the result type to be always printed.

```
Test Printing Synth.
```

This tells if the non-printing of synthesizable types is on or off. The default is to not print synthesizable types.

Printing matching on irrefutable pattern

If an inductive type has just one constructor, pattern-matching can be written using `let` ... `:=` ... `in` ...

```
Add Printing Let ident.
```

This adds *ident* to the list of inductive types for which pattern-matching is written using a `let` expression.

```
Remove Printing Let ident.
```

This removes *ident* from this list.

```
Test Printing Let for ident.
```

This tells if *ident* belongs to the list.

```
Print Table Printing Let.
```

This prints the list of inductive types for which pattern-matching is written using a `let` expression.

The list of inductive types for which pattern-matching is written using a `let` expression is managed synchronously. This means that it is sensible to the command `Reset`.

Printing matching on booleans

If an inductive type is isomorphic to the boolean type, pattern-matching can be written using `if` ... `then` ... `else` ...

```
Add Printing If ident.
```

This adds *ident* to the list of inductive types for which pattern-matching is written using an `if` expression.

```
Remove Printing If ident.
```

This removes *ident* from this list.

```
Test Printing If for ident.
```

This tells if *ident* belongs to the list.

```
Print Table Printing If.
```

This prints the list of inductive types for which pattern-matching is written using an `if` expression.

The list of inductive types for which pattern-matching is written using an `if` expression is managed synchronously. This means that it is sensible to the command `Reset`.

Example

This example emphasizes what the printing options offer.

```
Coq < Test Printing Let for prod.
Cases on elements of prod are printed using a 'let' form

Coq < Print fst.
fst =
fun (A B : Set) (p : A * B) => let 'pair x _ := p in x
     : forall A B : Set, A * B -> A
Argument scopes are [type_scope type_scope _]

Coq < Remove Printing Let prod.

Coq < Unset Printing Synth.

Coq < Unset Printing Wildcard.

Coq < Print fst.
fst =
fun (A B : Set) (p : A * B) => let 'pair x a := p return A in x
     : forall A B : Set, A * B -> A
Argument scopes are [type_scope type_scope _]
```

2.3 Advanced recursive functions

The *experimental* command

Function *ident* *binder*$_1$ ... *binder*$_n$ {decrease_annot} : type$_0$:= *term*$_0$

can be seen as a generalization of `Fixpoint`. It is actually a wrapper for several ways of defining a function *and other useful related objects*, namely: an induction principle that reflects the recursive structure of the function (see 8.7.7), and its fixpoint equality. The meaning of this declaration is to define a function *ident*, similarly to `Fixpoint`. Like in `Fixpoint`, the decreasing argument must be given (unless the function is not recursive), but it must not necessary be *structurally* decreasing. The point of the `{}` annotation is to name the decreasing argument *and* to describe which kind of decreasing criteria must be used to ensure termination of recursive calls.

The `Function` construction enjoys also the `with` extension to define mutually recursive definitions. However, this feature does not work for non structural recursive functions.

See the documentation of `functional induction` (see Section 8.7.7) and `Functional Scheme` (see Section 8.15 and 10.4) for how to use the induction principle to easily reason about the function.

Remark: To obtain the right principle, it is better to put rigid parameters of the function as first arguments. For example it is better to define plus like this:

```
Coq < Function plus (m n : nat) {struct n} : nat :=
Coq <   match n with
Coq <   | 0 => m
Coq <   | S p => S (plus m p)
Coq <   end.
```

than like this:

```
Coq < Function plus (n m : nat) {struct n} : nat :=
Coq <   match n with
Coq <   | 0 => m
Coq <   | S p => S (plus p m)
Coq <   end.
```

Limitations $term_0$ must be build as a *pure pattern-matching tree* (`match...with`) with applications only *at the end* of each branch. For now dependent cases are not treated.

Error messages:

1. `The recursive argument must be specified`
2. `No argument name` *ident*
3. `Cannot use mutual definition with well-founded recursion or measure`
4. `Cannot define graph for` *ident*`...` (warning)

   The generation of the graph relation `(R_`*ident*`)` used to compute the induction scheme of *ident* raised a typing error. Only the ident is defined, the induction scheme will not be generated.

   This error happens generally when:

   - the definition uses pattern matching on dependent types, which `Function` cannot deal with yet.
   - the definition is not a *pattern-matching tree* as explained above.

5. `Cannot define principle(s) for` *ident*`...` (warning)

   The generation of the graph relation `(R_`*ident*`)` succeeded but the induction principle could not be built. Only the ident is defined. Please report.

6. `Cannot build functional inversion principle` (warning)

   `functional inversion` will not be available for the function.

See also: 8.15, 10.4, 8.7.7

Depending on the `{...}` annotation, different definition mechanisms are used by `Function`. More precise description given below.

Variants:

1. `Function` *ident* $binder_1 \ldots binder_n$ `:` $type_0$ `:=` $term_0$

   Defines the not recursive function *ident* as if declared with `Definition`. Moreover the following are defined:

   - *ident*`_rect`, *ident*`_rec` and *ident*`_ind`, which reflect the pattern matching structure of $term_0$ (see the documentation of `Inductive` 1.3.3);
   - The inductive `R_`*ident* corresponding to the graph of *ident* (silently);
   - *ident*`_complete` and *ident*`_correct` which are inversion information linking the function and its graph.

2. `Function` *ident* *binder*$_1$ ... *binder*$_n$ `{struct` *ident*$_0$`}` `:` type$_0$ `:=` *term*$_0$

   Defines the structural recursive function *ident* as if declared with `Fixpoint`. Moreover the following are defined:

   - The same objects as above;
   - The fixpoint equation of *ident*: *ident*`_equation`.

3. `Function` *ident* *binder*$_1$ ... *binder*$_n$ `{measure` *term*$_1$ *ident*$_0$`}` `:` type$_0$ `:=` *term*$_0$

4. `Function` *ident* *binder*$_1$ ... *binder*$_n$ `{wf` *term*$_1$ *ident*$_0$`}` `:` type$_0$ `:=` *term*$_0$

   Defines a recursive function by well founded recursion. **The module `Recdef` of the standard library must be loaded for this feature**. The `{}` annotation is mandatory and must be one of the following:

   - `{measure` *term*$_1$ *ident*$_0$`}` with *ident*$_0$ being the decreasing argument and *term*$_1$ being a function from type of *ident*$_0$ to `nat` for which value on the decreasing argument decreases (for the `lt` order on `nat`) at each recursive call of *term*$_0$, parameters of the function are bound in *term*$_0$;
   - `{wf` *term*$_1$ *ident*$_0$`}` with *ident*$_0$ being the decreasing argument and *term*$_1$ an ordering relation on the type of *ident*$_0$ (i.e. of type $\mathrm{T}_{ident_0} \to \mathrm{T}_{ident_0} \to$ `Prop`) for which the decreasing argument decreases at each recursive call of *term*$_0$. The order must be well founded. parameters of the function are bound in *term*$_0$.

   Depending on the annotation, the user is left with some proof obligations that will be used to define the function. These proofs are: proofs that each recursive call is actually decreasing with respect to the given criteria, and (if the criteria is `wf`) a proof that the ordering relation is well founded.

   Once proof obligations are discharged, the following objects are defined:

   - The same objects as with the `struct`;
   - The lemma *ident*`_tcc` which collects all proof obligations in one property;
   - The lemmas *ident*`_terminate` and *ident*`_F` which is needed to be inlined during extraction of *ident*.

   The way this recursive function is defined is the subject of several papers by Yves Bertot and Antonia Balaa on one hand and Gilles Barthe, Julien Forest, David Pichardie and Vlad Rusu on the other hand.

   Remark: Proof obligations are presented as several subgoals belonging to a Lemma *ident*`_tcc`.

2.4 Section mechanism

The sectioning mechanism allows to organize a proof in structured sections. Then local declarations become available (see Section 1.3.2).

2.4.1 Section *ident*

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

2.4.2 End *ident*

This command closes the section named *ident*. When a section is closed, all local declarations (variables and local definitions) are *discharged*. This means that all global objects defined in the section are generalized with respect to all variables and local definitions it depends on in the section. None of the local declarations (considered as autonomous declarations) survive the end of the section.

Here is an example :

```
Coq < Section s1.

Coq < Variables x y : nat.
x is assumed
y is assumed

Coq < Let y' := y.
y' is defined

Coq < Definition x' := S x.
x' is defined

Coq < Definition x'' := x' + y'.
x'' is defined

Coq < Print x'.
x' = S x
     : nat

Coq < End s1.

Coq < Print x'.
x' = fun x : nat => S x
     : nat -> nat
Argument scope is [nat_scope]

Coq < Print x''.
x'' = fun x y : nat => let y' := y in x' x + y'
     : nat -> nat -> nat
Argument scopes are [nat_scope nat_scope]
```

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

Error messages:

1. `This is not the last opened section`

Remarks:

1. Most commands, like `Hint`, `Notation`, option management, ... which appear inside a section are canceled when the section is closed.

2.5 Module system

The module system provides a way of packaging related elements together, as well as a mean of massive abstraction.

module_type	::=	*qualid*
	|	*module_type* `with Definition` *qualid* := *term*
	|	*module_type* `with Module` *qualid* := *qualid*
	|	*qualid qualid* ... *qualid*
module_binding	::=	`(` `[Import|Export]` *ident* ... *ident* `:` *module_type* `)`
module_bindings	::=	*module_binding* ... *module_binding*
module_expression	::=	*qualid* ... *qualid*

Figure 2.3: Syntax of modules

2.5.1 `Module` *ident*

This command is used to start an interactive module named *ident*.

Variants:

1. `Module` *ident module_bindings*

   Starts an interactive functor with parameters given by *module_bindings*.

2. `Module` *ident* `:` *module_type*

   Starts an interactive module specifying its module type.

3. `Module` *ident module_bindings* `:` *module_type*

   Starts an interactive functor with parameters given by *module_bindings*, and output module type *module_type*.

4. `Module` *ident* `<:` *module_type*

   Starts an interactive module satisfying *module_type*.

5. `Module` *ident module_bindings* `<:` *module_type*

   Starts an interactive functor with parameters given by *module_bindings*. The output module type is verified against the module type *module_type*.

6. `Module [Import|Export]`

   Behaves like `Module`, but automatically imports or exports the module.

Reserved commands inside an interactive module:

1. `Include` *module_expression*

   Includes the content of *module_expression* in the current interactive module.

2. `Include Type` *module_type*

   Includes the content of *module_type* in the current interactive module.

2.5.2 `End` *ident*

This command closes the interactive module *ident*. If the module type was given the content of the module is matched against it and an error is signaled if the matching fails. If the module is basic (is not a functor) its components (constants, inductive types, submodules etc) are now available through the dot notation.

Error messages:

1. `No such label` *ident*
2. `Signature components for label` *ident* `do not match`
3. `This is not the last opened module`

2.5.3 `Module` *ident* `:=` *module_expression*

This command defines the module identifier *ident* to be equal to *module_expression*.

Variants:

1. `Module` *ident* *module_bindings* `:=` *module_expression*

   Defines a functor with parameters given by *module_bindings* and body *module_expression*.

2. `Module` *ident* *module_bindings* `:` *module_type* `:=` *module_expression*

   Defines a functor with parameters given by *module_bindings* (possibly none), and output module type *module_type*, with body *module_expression*.

3. `Module` *ident* *module_bindings* `<:` *module_type* `:=` *module_expression*

   Defines a functor with parameters given by *module_bindings* (possibly none) with body *module_expression*. The body is checked against *module_type*.

2.5.4 `Module Type` *ident*

This command is used to start an interactive module type *ident*.

Variants:

1. `Module Type` *ident* *module_bindings*

   Starts an interactive functor type with parameters given by *module_bindings*.

Reserved commands inside an interactive module type:

1. `Include` *module_expression*

   Includes the content of *module_expression* in the current interactive module type.

2. `Include Type` *module_type*

   Includes the content of *module_type* in the current interactive module type.

3. *declaration_keyword* `Inline` *assums*

   This declaration will be automatically unfolded at functor application.

2.5.5 End *ident*

This command closes the interactive module type *ident*.

Error messages:

1. `This is not the last opened module type`

2.5.6 Module Type *ident* := *module_type*

Defines a module type *ident* equal to *module_type*.

Variants:

1. Module Type *ident module_bindings* := *module_type*

   Defines a functor type *ident* specifying functors taking arguments *module_bindings* and returning *module_type*.

2.5.7 Declare Module *ident* : *module_type*

Declares a module *ident* of type *module_type*.

Variants:

1. Declare Module *ident module_bindings* : *module_type*

   Declares a functor with parameters *module_bindings* and output module type *module_type*.

Example

Let us define a simple module.

```
Coq < Module M.
Interactive Module M started

Coq <   Definition T := nat.
T is defined

Coq <   Definition x := 0.
x is defined

Coq <   Definition y : bool.
1 subgoal

  ============================
   bool

Coq <     exact true.
Proof completed.

Coq <   Defined.
exact true.
y is defined

Coq < End M.
Module M is defined
```

Inside a module one can define constants, prove theorems and do any other things that can be done in the toplevel. Components of a closed module can be accessed using the dot notation:

```
Coq < Print M.x.
M.x = 0
     : nat
```

A simple module type:

```
Coq < Module Type SIG.
Interactive Module Type SIG started

Coq <   Parameter T : Set.
T is assumed

Coq <   Parameter x : T.
x is assumed

Coq < End SIG.
Module Type SIG is defined
```

Inside a module type the proof editing mode is not available. Consequently commands like `Definition` without body, `Lemma`, `Theorem` are not allowed. In order to declare constants, use `Axiom` and `Parameter`.

Now we can create a new module from `M`, giving it a less precise specification: the `y` component is dropped as well as the body of `x`.

```
Coq < Module N  :  SIG with Definition T := nat  :=  M.
Coq < Coq < Module N is defined

Coq < Print N.T.
N.T = nat
     : Set

Coq < Print N.x.
*** [ N.x : N.T ]

Coq < Print N.y.
Error: N.y not a defined object.
```

The definition of `N` using the module type expression `SIG with Definition T:=nat` is equivalent to the following one:

```
Coq < Module Type SIG'.

Coq <   Definition T : Set := nat.

Coq <   Parameter x : T.

Coq < End SIG'.

Coq < Module N : SIG' := M.
```

If we just want to be sure that the our implementation satisfies a given module type without restricting the interface, we can use a transparent constraint

```
Coq < Module P <: SIG := M.
Module P is defined

Coq < Print P.y.
M.y = true
     : bool
```

Now let us create a functor, i.e. a parametric module

```
Coq < Module Two (X Y: SIG).
Interactive Module Two started

Coq <   Definition T := (X.T * Y.T)%type.

Coq <   Definition x := (X.x, Y.x).

Coq < End Two.
Module Two is defined
```

and apply it to our modules and do some computations

```
Coq < Module Q := Two M N.
Module Q is defined

Coq < Eval compute in (fst Q.x + snd Q.x).
     = N.x
     : nat
```

In the end, let us define a module type with two sub-modules, sharing some of the fields and give one of its possible implementations:

```
Coq < Module Type SIG2.
Interactive Module Type SIG2 started

Coq <   Declare Module M1 : SIG.
Module M1 is declared

Coq <   Module M2 <: SIG.
Interactive Module M2 started

Coq <     Definition T := M1.T.
T is defined

Coq <     Parameter x : T.
x is assumed

Coq <   End M2.
Module M2 is defined

Coq < End SIG2.
Module Type SIG2 is defined

Coq < Module Mod <: SIG2.

Coq <   Module M1.

Coq <     Definition T := nat.

Coq <     Definition x := 1.

Coq <   End M1.

Coq <   Module M2 := M.

Coq < End Mod.
Module Mod is defined
```

Notice that `M` is a correct body for the component `M2` since its `T` component is equal `nat` and hence `M1.T` as specified.

Remarks:

1. Modules and module types can be nested components of each other.
2. When a module declaration is started inside a module type, the proof editing mode is still unavailable.
3. One can have sections inside a module or a module type, but not a module or a module type inside a section.
4. Commands like `Hint` or `Notation` can also appear inside modules and module types. Note that in case of a module definition like:

   ```
   Module N : SIG := M.
   ```

   or

   ```
   Module N : SIG.
   ...
   End N.
   ```

   hints and the like valid for `N` are not those defined in `M` (or the module body) but the ones defined in `SIG`.

2.5.8 `Import` *qualid*

If *qualid* denotes a valid basic module (i.e. its module type is a signature), makes its components available by their short names.

Example:

```
Coq < Module Mod.
Interactive Module Mod started

Coq <   Definition T:=nat.
T is defined

Coq <   Check T.
T
     : Set

Coq < End Mod.
Module Mod is defined

Coq < Check Mod.T.
Mod.T
     : Set

Coq < Check T. (* Incorrect ! *)
Toplevel input, characters 6-7:
> Check T.
>       ^
Error: The reference T was not found in the current environment.

Coq < Import Mod.

Coq < Check T. (* Now correct *)
T
     : Set
```

Some features defined in modules are activated only when a module is imported. This is for instance the case of notations (see Section 12.1).

Variants:

1. `Export` *qualid*

   When the module containing the command `Export` *qualid* is imported, *qualid* is imported as well.

Error messages:

1. *qualid* `is not a module`

Warnings:

1. Warning: Trying to mask the absolute name *qualid* !

2.5.9 `Print Module` *ident*

Prints the module type and (optionally) the body of the module *ident*.

2.5.10 `Print Module Type` *ident*

Prints the module type corresponding to *ident*.

2.5.11 `Locate Module` *qualid*

Prints the full name of the module *qualid*.

2.6 Libraries and qualified names

2.6.1 Names of libraries and files

Libraries The theories developed in COQ are stored in *library files* which are hierarchically classified into *libraries* and *sublibraries*. To express this hierarchy, library names are represented by qualified identifiers *qualid*, i.e. as list of identifiers separated by dots (see Section 1.2.3). For instance, the library file `Mult` of the standard COQ library `Arith` has name `Coq.Arith.Mult`. The identifier that starts the name of a library is called a *library root*. All library files of the standard library of COQ have reserved root `Coq` but library file names based on other roots can be obtained by using `coqc` options `-I` or `-R` (see Section 13.5). Also, when an interactive COQ session starts, a library of root `Top` is started, unless option `-top` or `-notop` is set (see Section 13.5).

As library files are stored on the file system of the underlying operating system, a translation from file-system names to COQ names is needed. In this translation, names in the file system are called *physical* paths while COQ names are contrastingly called *logical* names. Logical names are mapped to physical paths using the commands `Add LoadPath` or `Add Rec LoadPath` (see Sections 6.5.3 and 6.5.4).

2.6.2 Qualified names

Library files are modules which possibly contain submodules which eventually contain constructions (axioms, parameters, definitions, lemmas, theorems, remarks or facts). The *absolute name*, or *full name*, of a construction in some library file is a qualified identifier starting with the logical name of the library file, followed by the sequence of submodules names encapsulating the construction and ended by the proper name of the construction. Typically, the absolute name `Coq.Init.Logic.eq` denotes Leibniz' equality defined in the module `Logic` in the sublibrary `Init` of the standard library of Coq.

The proper name that ends the name of a construction is the *short name* (or sometimes *base name*) of the construction (for instance, the short name of `Coq.Init.Logic.eq` is `eq`). Any partial suffix of the absolute name is a *partially qualified name* (e.g. `Logic.eq` is a partially qualified name for `Coq.Init.Logic.eq`). Especially, the short name of a construction is its shortest partially qualified name.

Coq does not accept two constructions (definition, theorem, ...) with the same absolute name but different constructions can have the same short name (or even same partially qualified names as soon as the full names are different).

Notice that the notion of absolute, partially qualified and short names also applies to library file names.

Visibility Coq maintains a table called *name table* which maps partially qualified names of constructions to absolute names. This table is updated by the commands `Require` (see 6.4.1), `Import` and `Export` (see 2.5.8) and also each time a new declaration is added to the context. An absolute name is called *visible* from a given short or partially qualified name when this latter name is enough to denote it. This means that the short or partially qualified name is mapped to the absolute name in Coq name table.

A similar table exists for library file names. It is updated by the vernacular commands `Add LoadPath` and `Add Rec LoadPath` (or their equivalent as options of the Coq executables, `-I` and `-R`).

It may happen that a visible name is hidden by the short name or a qualified name of another construction. In this case, the name that has been hidden must be referred to using one more level of qualification. To ensure that a construction always remains accessible, absolute names can never be hidden.

Examples:

```
Coq < Check 0.
0
     : nat

Coq < Definition nat := bool.
nat is defined

Coq < Check 0.
0
     : Datatypes.nat

Coq < Check Datatypes.nat.
Datatypes.nat
     : Set

Coq < Locate nat.
Constant Top.nat
Inductive Coq.Init.Datatypes.nat
```

```
(shorter name to refer to it in current context is Datatypes.nat)
```

See also: Command `Locate` in Section 6.2.9 and `Locate Library` in Section 6.5.11.

2.7 Implicit arguments

An implicit argument of a function is an argument which can be inferred from contextual knowledge. There are different kinds of implicit arguments that can be considered implicit in different ways. There are also various commands to control the setting or the inference of implicit arguments.

2.7.1 The different kinds of implicit arguments

Implicit arguments inferable from the knowledge of other arguments of a function

The first kind of implicit arguments covers the arguments that are inferable from the knowledge of the type of other arguments of the function, or of the type of the surrounding context of the application. Especially, such implicit arguments correspond to parameters dependent in the type of the function. Typical implicit arguments are the type arguments in polymorphic functions. There are several kinds of such implicit arguments.

Strict Implicit Arguments. An implicit argument can be either strict or non strict. An implicit argument is said *strict* if, whatever the other arguments of the function are, it is still inferable from the type of some other argument. Technically, an implicit argument is strict if it corresponds to a parameter which is not applied to a variable which itself is another parameter of the function (since this parameter may erase its arguments), not in the body of a `match`, and not itself applied or matched against patterns (since the original form of the argument can be lost by reduction).

For instance, the first argument of

```
cons: forall A:Set, A -> list A -> list A
```

in module `List.v` is strict because `list` is an inductive type and `A` will always be inferable from the type `list A` of the third argument of `cons`. On the contrary, the second argument of a term of type

```
forall P:nat->Prop, forall n:nat, P n -> ex nat P
```

is implicit but not strict, since it can only be inferred from the type `P n` of the third argument and if `P` is, e.g., `fun _ => True`, it reduces to an expression where `n` does not occur any longer. The first argument `P` is implicit but not strict either because it can only be inferred from `P n` and `P` is not canonically inferable from an arbitrary `n` and the normal form of `P n` (consider e.g. that `n` is `0` and the third argument has type `True`, then any `P` of the form `fun n => match n with 0 => True | _ => anything end` would be a solution of the inference problem).

Contextual Implicit Arguments. An implicit argument can be *contextual* or not. An implicit argument is said *contextual* if it can be inferred only from the knowledge of the type of the context of the current expression. For instance, the only argument of

```
nil : forall A:Set, list A
```

is contextual. Similarly, both arguments of a term of type

```
forall P:nat->Prop, forall n:nat, P n \/ n = 0
```

are contextual (moreover, `n` is strict and `P` is not).

Reversible-Pattern Implicit Arguments. There is another class of implicit arguments that can be reinferred unambiguously if all the types of the remaining arguments are known. This is the class of implicit arguments occurring in the type of another argument in position of reversible pattern, which means it is at the head of an application but applied only to uninstantiated distinct variables. Such an implicit argument is called *reversible-pattern implicit argument*. A typical example is the argument `P` of `nat_rec` in

```
nat_rec : forall P : nat -> Set, P 0 -> (forall n : nat, P
n -> P (S n)) -> forall x : nat, P x.
```

(`P` is reinferable by abstracting over `n` in the type `P n`).

See Section 2.7.9 for the automatic declaration of reversible-pattern implicit arguments.

Implicit arguments inferable by resolution

This corresponds to a class of non dependent implicit arguments that are solved based on the structure of their type only.

2.7.2 Maximal or non maximal insertion of implicit arguments

In case a function is partially applied, and the next argument to be applied is an implicit argument, two disciplines are applicable. In the first case, the function is considered to have no arguments furtherly: one says that the implicit argument is not maximally inserted. In the second case, the function is considered to be implicitly applied to the implicit arguments it is waiting for: one says that the implicit argument is maximally inserted.

Each implicit argument can be declared to have to be inserted maximally or non maximally. This can be governed argument per argument by the command `Implicit Arguments` (see 2.7.4) or globally by the command `Set Maximal Implicit Insertion` (see 2.7.10). See also Section 2.7.12.

2.7.3 Casual use of implicit arguments

In a given expression, if it is clear that some argument of a function can be inferred from the type of the other arguments, the user can force the given argument to be guessed by replacing it by "_". If possible, the correct argument will be automatically generated.

Error messages:

1. `Cannot infer a term for this placeholder`

   Coq was not able to deduce an instantiation of a "_".

2.7.4 Declaration of implicit arguments for a constant

In case one wants that some arguments of a given object (constant, inductive types, constructors, assumptions, local or not) are always inferred by Coq, one may declare once and for all which are the expected implicit arguments of this object. There are two ways to do this, a-priori and a-posteriori.

Implicit Argument Binders

In the first setting, one wants to explicitely give the implicit arguments of a constant as part of its definition. To do this, one has to surround the bindings of implicit arguments by curly braces:

```
Coq < Definition id {A : Type} (x : A) : A := x.
id is defined
```

This automatically declares the argument `A` of `id` as a maximally inserted implicit argument. One can then do as-if the argument was absent in every situation but still be able to specify it if needed:

```
Coq < Definition compose {A B C} (g : B -> C) (f : A -> B) :=
Coq <   fun x => g (f x).
compose is defined

Coq < Goal forall A, compose id id = id (A:=A).
1 subgoal

  ============================
   forall A : Type, compose id id = id
```

The syntax is supported in all top-level definitions: `Definition`, `Fixpoint`, `Lemma` and so on. For (co-)inductive datatype declarations, the semantics are the following: an inductive parameter declared as an implicit argument need not be repeated in the inductive definition but will become implicit for the constructors of the inductive only, not the inductive type itself. For example:

```
Coq < Inductive list {A : Type} : Type :=
Coq < | nil : list
Coq < | cons : A -> list -> list.
list is defined
list_rect is defined
list_ind is defined
list_rec is defined

Coq < Print list.
Inductive list (A : Type) : Type :=  nil : list | cons : A -> list -> list
For list: Argument A is implicit and maximally inserted
For nil: Argument A is implicit and maximally inserted
For cons: Argument A is implicit and maximally inserted
For list: Argument scope is [type_scope]
For nil: Argument scope is [type_scope]
For cons: Argument scopes are [type_scope _ _]
```

One can always specify the parameter if it is not uniform using the usual implicit arguments disambiguation syntax.

The Implicit Arguments Vernacular Command

To set implicit arguments for a constant a-posteriori, one can use the command:

`Implicit Arguments` *qualid* `[` *possibly_bracketed_ident* `...` *possibly_bracketed_ident* `]`

where the list of *possibly_bracketed_ident* is the list of parameters to be declared implicit, each of the identifier of the list being optionally surrounded by square brackets, then meaning that this parameter has to be maximally inserted.

After the above declaration is issued, implicit arguments can just (and have to) be skipped in any expression involving an application of *qualid*.

Variants:

1. `Global Implicit Arguments` *qualid* `[` *possibly_bracketed_ident* `...` *possibly_bracketed_ident* `]`

   Tells to recompute the implicit arguments of *qualid* after ending of the current section if any, enforcing the implicit arguments known from inside the section to be the ones declared by the command.

2. `Local Implicit Arguments` *qualid* `[` *possibly_bracketed_ident* `...` *possibly_bracketed_ident* `]`

   When in a module, tells not to activate the implicit arguments of *qualid* declared by this commands to contexts that requires the module.

Example:

```
Coq < Inductive list (A:Type) : Type :=
Coq <   | nil : list A
Coq <   | cons : A -> list A -> list A.

Coq < Check (cons nat 3 (nil nat)).
cons nat 3 (nil nat)
     : list nat

Coq < Implicit Arguments cons [A].

Coq < Implicit Arguments nil [A].

Coq < Check (cons 3 nil).
cons 3 nil
     : list nat

Coq < Fixpoint map (A B:Type) (f:A->B) (l:list A) : list B :=
Coq <   match l with nil => nil | cons a t => cons (f a) (map A B f t) end.
map is recursively defined (decreasing on 4th argument)

Coq < Fixpoint length (A:Type) (l:list A) : nat :=
Coq <   match l with nil => 0 | cons _ m => S (length A m) end.
length is recursively defined (decreasing on 2nd argument)

Coq < Implicit Arguments map [A B].

Coq < Implicit Arguments length [[A]]. (* A has to be maximally inserted *)

Coq < Check (fun l:list (list nat) => map length l).
fun l : list (list nat) => map length l
     : list (list nat) -> list nat
```

Remark: To know which are the implicit arguments of an object, use the command `Print Implicit` (see 2.7.12).

Remark: If the list of arguments is empty, the command removes the implicit arguments of *qualid*.

2.7.5 Automatic declaration of implicit arguments for a constant

COQ can also automatically detect what are the implicit arguments of a defined object. The command is just

`Implicit Arguments` *qualid*

The auto-detection is governed by options telling if strict, contextual, or reversible-pattern implicit arguments must be considered or not (see Sections 2.7.7, 2.7.8, 2.7.9 and also 2.7.10).

Variants:

1. `Global Implicit Arguments` *qualid*

   Tells to recompute the implicit arguments of *qualid* after ending of the current section if any.

2. `Local Implicit Arguments` *qualid*

   When in a module, tells not to activate the implicit arguments of *qualid* computed by this declaration to contexts that requires the module.

Example:

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

Coq < Implicit Arguments cons.

Coq < Print Implicit cons.
cons : forall A : Set, A -> list A -> list A
Argument A is implicit

Coq < Implicit Arguments nil.

Coq < Print Implicit nil.
nil : forall A : Set, list A
No implicit arguments

Coq < Set Contextual Implicit.

Coq < Implicit Arguments nil.

Coq < Print Implicit nil.
nil : forall A : Set, list A
Argument A is implicit and maximally inserted
```

The computation of implicit arguments takes account of the unfolding of constants. For instance, the variable `p` below has type `(Transitivity R)` which is reducible to `forall x,y:U, R x y -> forall z:U, R y z -> R x z`. As the variables `x`, `y` and `z` appear strictly in body of the type, they are implicit.

```
Coq < Variable X : Type.

Coq < Definition Relation := X -> X -> Prop.

Coq < Definition Transitivity (R:Relation) :=
Coq <   forall x y:X, R x y -> forall z:X, R y z -> R x z.

Coq < Variables (R : Relation) (p : Transitivity R).
```

```
Coq < Implicit Arguments p.

Coq < Print p.
*** [ p : Transitivity R ]
Expanded type for implicit arguments
p : forall x y : X, R x y -> forall z : X, R y z -> R x z
Arguments x, y, z are implicit

Coq < Print Implicit p.
p : forall x y : X, R x y -> forall z : X, R y z -> R x z
Arguments x, y, z are implicit

Coq < Variables (a b c : X) (r1 : R a b) (r2 : R b c).

Coq < Check (p r1 r2).
p r1 r2
     : R a c
```

2.7.6 Mode for automatic declaration of implicit arguments

In case one wants to systematically declare implicit the arguments detectable as such, one may switch to the automatic declaration of implicit arguments mode by using the command

```
Set Implicit Arguments.
```

Conversely, one may unset the mode by using `Unset Implicit Arguments`. The mode is off by default. Auto-detection of implicit arguments is governed by options controlling whether strict and contextual implicit arguments have to be considered or not.

2.7.7 Controlling strict implicit arguments

When the mode for automatic declaration of implicit arguments is on, the default is to automatically set implicit only the strict implicit arguments plus, for historical reasons, a small subset of the non strict implicit arguments. To relax this constraint and to set implicit all non strict implicit arguments by default, use the command

```
Unset Strict Implicit.
```

Conversely, use the command `Set Strict Implicit` to restore the original mode that declares implicit only the strict implicit arguments plus a small subset of the non strict implicit arguments.

In the other way round, to capture exactly the strict implicit arguments and no more than the strict implicit arguments, use the command:

```
Set Strongly Strict Implicit.
```

Conversely, use the command `Unset Strongly Strict Implicit` to let the option "`Strict Implicit`" decide what to do.

Remark: In versions of Coq prior to version 8.0, the default was to declare the strict implicit arguments as implicit.

term	++=	@ *qualid term ... term*
	|	@ *qualid*
	|	*qualid argument ... argument*
argument	::=	*term*
	|	(*ident* := *term*)

Figure 2.4: Syntax for explicitly giving implicit arguments

2.7.8 Controlling contextual implicit arguments

By default, COQ does not automatically set implicit the contextual implicit arguments. To tell COQ to infer also contextual implicit argument, use command

```
Set Contextual Implicit.
```

Conversely, use command `Unset Contextual Implicit` to unset the contextual implicit mode.

2.7.9 Controlling reversible-pattern implicit arguments

By default, COQ does not automatically set implicit the reversible-pattern implicit arguments. To tell COQ to infer also reversible-pattern implicit argument, use command

```
Set Reversible Pattern Implicit.
```

Conversely, use command `Unset Reversible Pattern Implicit` to unset the reversible-pattern implicit mode.

2.7.10 Controlling the insertion of implicit arguments not followed by explicit arguments

Implicit arguments can be declared to be automatically inserted when a function is partially applied and the next argument of the function is an implicit one. In case the implicit arguments are automatically declared (with the command `Set Implicit Arguments`), the command

```
Set Maximal Implicit Insertion.
```

is used to tell to declare the implicit arguments with a maximal insertion status. By default, automatically declared implicit arguments are not declared to be insertable maximally. To restore the default mode for maximal insertion, use command `Unset Maximal Implicit Insertion`.

2.7.11 Explicit applications

In presence of non strict or contextual argument, or in presence of partial applications, the synthesis of implicit arguments may fail, so one may have to give explicitly certain implicit arguments of an application. The syntax for this is (*ident* := *term*) where *ident* is the name of the implicit argument and *term* is its corresponding explicit term. Alternatively, one can locally deactivate the hiding of implicit arguments of a function by using the notation @*qualid* $term_1 \ldots term_n$. This syntax extension is given Figure 2.4.

Example (continued):

```
Coq < Check (p r1 (z:=c)).
p r1 (z:=c)
     : R b c -> R a c

Coq < Check (p (x:=a) (y:=b) r1 (z:=c) r2).
p r1 r2
     : R a c
```

2.7.12 Displaying what the implicit arguments are

To display the implicit arguments associated to an object, and to know if each of them is to be used maximally or not, use the command

`Print Implicit` *qualid* .

2.7.13 Explicit displaying of implicit arguments for pretty-printing

By default the basic pretty-printing rules hide the inferable implicit arguments of an application. To force printing all implicit arguments, use command

```
Set Printing Implicit.
```

Conversely, to restore the hiding of implicit arguments, use command

```
Unset Printing Implicit.
```

By default the basic pretty-printing rules display the implicit arguments that are not detected as strict implicit arguments. This "defensive" mode can quickly make the display cumbersome so this can be deactivated by using the command

```
Unset Printing Implicit Defensive.
```

Conversely, to force the display of non strict arguments, use command

```
Set Printing Implicit Defensive.
```

See also: `Set Printing All` in Section 2.9.

2.7.14 Interaction with subtyping

When an implicit argument can be inferred from the type of more than one of the other arguments, then only the type of the first of these arguments is taken into account, and not an upper type of all of them. As a consequence, the inference of the implicit argument of "=" fails in

```
Coq < Check nat = Prop.
```

but succeeds in

```
Coq < Check Prop = nat.
```

2.7.15 Canonical structures

A canonical structure is an instance of a record/structure type that can be used to solve equations involving implicit arguments. Assume that *qualid* denotes an object $(Build\_struc\ c_1\ \ldots\ c_n)$ in the structure *struct* of which the fields are x_1, ..., x_n. Assume that *qualid* is declared as a canonical structure using the command

```
Canonical Structure qualid.
```

Then, each time an equation of the form $(x_i\ \_) =_{\beta\delta\iota\zeta} c_i$ has to be solved during the type-checking process, *qualid* is used as a solution. Otherwise said, *qualid* is canonically used to extend the field c_i into a complete structure built on c_i.

Canonical structures are particularly useful when mixed with coercions and strict implicit arguments. Here is an example.

```
Coq < Require Import Relations.
Coq < Require Import EqNat.
Coq < Set Implicit Arguments.
Coq < Unset Strict Implicit.
Coq < Structure Setoid : Type :=
Coq <   {Carrier :> Set;
Coq <    Equal : relation Carrier;
Coq <    Prf_equiv : equivalence Carrier Equal}.
Coq < Definition is_law (A B:Setoid) (f:A -> B) :=
Coq <   forall x y:A, Equal x y -> Equal (f x) (f y).
Coq < Axiom eq_nat_equiv : equivalence nat eq_nat.
Coq < Definition nat_setoid : Setoid := Build_Setoid eq_nat_equiv.
Coq < Canonical Structure nat_setoid.
```

Thanks to `nat_setoid` declared as canonical, the implicit arguments `A` and `B` can be synthesized in the next statement.

```
Coq < Lemma is_law_S : is_law S.
1 subgoal

  ============================
   is_law (A:=nat_setoid) (B:=nat_setoid) S
```

Remark: If a same field occurs in several canonical structure, then only the structure declared first as canonical is considered.

Variants:

1. `Canonical Structure` *ident* `:=` *term* `:` *type*`.`
   `Canonical Structure` *ident* `:=` *term*`.`
   `Canonical Structure` *ident* `:` *type* `:=` *term*`.`

   These are equivalent to a regular definition of *ident* followed by the declaration

   `Canonical Structure` *ident*.

See also: more examples in user contribution `category` (`Rocq/ALGEBRA`).

Print Canonical Projections.

This displays the list of global names that are components of some canonical structure. For each of them, the canonical structure of which it is a projection is indicated. For instance, the above example gives the following output:

```
Coq < Print Canonical Projections.
eq_nat_equiv <- Prf_equiv ( nat_setoid )
eq_nat <- Equal ( nat_setoid )
nat <- Carrier ( nat_setoid )
```

2.7.16 Implicit types of variables

It is possible to bind variable names to a given type (e.g. in a development using arithmetic, it may be convenient to bind the names `n` or `m` to the type `nat` of natural numbers). The command for that is

`Implicit Types` *ident* ... *ident* : *type*

The effect of the command is to automatically set the type of bound variables starting with *ident* (either *ident* itself or *ident* followed by one or more single quotes, underscore or digits) to be *type* (unless the bound variable is already declared with an explicit type in which case, this latter type is considered).

Example:

```
Coq < Require Import List.

Coq < Implicit Types m n : nat.

Coq < Lemma cons_inj_nat : forall m n l, n :: l = m :: l -> n = m.
1 subgoal

  ============================
   forall m n (l : list nat), n :: l = m :: l -> n = m

Coq < intros m n.
1 subgoal

  m : nat
  n : nat
  ============================
   forall l : list nat, n :: l = m :: l -> n = m

Coq < Lemma cons_inj_bool : forall (m n:bool) l, n :: l = m :: l -> n = m.
1 subgoal

  ============================
   forall (m n : bool) (l : list bool), n :: l = m :: l -> n = m
```

Variants:

1. `Implicit Type` *ident* : *type*
   This is useful for declaring the implicit type of a single variable.

2.8 Coercions

Coercions can be used to implicitly inject terms from one *class* in which they reside into another one. A *class* is either a sort (denoted by the keyword `Sortclass`), a product type (denoted by the keyword `Funclass`), or a type constructor (denoted by its name), e.g. an inductive type or any constant with a type of the form `forall` $(x_1 : A_1)..(x_n : A_n),\ s$ where s is a sort.

Then the user is able to apply an object that is not a function, but can be coerced to a function, and more generally to consider that a term of type A is of type B provided that there is a declared coercion between A and B. The main command is

`Coercion` *qualid* `:` $class_1$ `>->` $class_2$`.`

which declares the construction denoted by *qualid* as a coercion between $class_1$ and $class_2$.

More details and examples, and a description of the commands related to coercions are provided in Chapter 17.

2.9 Printing constructions in full

Coercions, implicit arguments, the type of pattern-matching, but also notations (see Chapter 12) can obfuscate the behavior of some tactics (typically the tactics applying to occurrences of subterms are sensitive to the implicit arguments). The command

```
Set Printing All.
```

deactivates all high-level printing features such as coercions, implicit arguments, returned type of pattern-matching, notations and various syntactic sugar for pattern-matching or record projections. Otherwise said, `Set Printing All` includes the effects of the commands `Set Printing Implicit`, `Set Printing Coercions`, `Set Printing Synth`, `Unset Printing Projections` and `Unset Printing Notations`. To reactivate the high-level printing features, use the command

```
Unset Printing All.
```

2.10 Printing universes

The following command:

```
Set Printing Universes
```

activates the display of the actual level of each occurrence of Type. See Section 4.1.1 for details. This wizard option, in combination with `Set Printing All` (see section 2.9) can help to diagnose failures to unify terms apparently identical but internally different in the Calculus of Inductive Constructions. To reactivate the display of the actual level of the occurrences of Type, use

```
Unset Printing Universes.
```

The constraints on the internal level of the occurrences of Type (see Section 4.1.1) can be printed using the command

```
Print Universes.
```

Chapter 3

The Coq library

The Coq library is structured into two parts:

The initial library: it contains elementary logical notions and data-types. It constitutes the basic state of the system directly available when running Coq;

The standard library: general-purpose libraries containing various developments of Coq axiomatizations about sets, lists, sorting, arithmetic, etc. This library comes with the system and its modules are directly accessible through the `Require` command (see Section 6.4.1);

In addition, user-provided libraries or developments are provided by Coq users' community. These libraries and developments are available for download at `http://coq.inria.fr` (see Section 3.3).

The chapter briefly reviews the Coq libraries.

3.1 The basic library

This section lists the basic notions and results which are directly available in the standard Coq system[1].

3.1.1 Notations

This module defines the parsing and pretty-printing of many symbols (infixes, prefixes, etc.). However, it does not assign a meaning to these notations. The purpose of this is to define and fix once for all the precedence and associativity of very common notations. The main notations fixed in the initial state are listed on Figure 3.1.

3.1.2 Logic

The basic library of Coq comes with the definitions of standard (intuitionistic) logical connectives (they are defined as inductive constructions). They are equipped with an appealing syntax enriching the (subclass *form*) of the syntactic class *term*. The syntax extension is shown on Figure 3.2.

Remark: Implication is not defined but primitive (it is a non-dependent product of a proposition over another proposition). There is also a primitive universal quantification (it is a dependent product over a

[1] Most of these constructions are defined in the `Prelude` module in directory `theories/Init` at the Coq root directory; this includes the modules `Notations`, `Logic`, `Datatypes`, `Specif`, `Peano`, `Wf` and `Tactics`. Module `Logic_Type` also makes it in the initial state

Notation	Precedence	Associativity
_ <-> _	95	no
_ \/ _	85	right
_ /\ _	80	right
~ _	75	right
_ = _	70	no
_ = _ = _	70	no
_ = _ :> _	70	no
_ <> _	70	no
_ <> _ :> _	70	no
_ < _	70	no
_ > _	70	no
_ <= _	70	no
_ >= _	70	no
_ < _ < _	70	no
_ < _ <= _	70	no
_ <= _ < _	70	no
_ <= _ <= _	70	no
_ + _	50	left
_ || _	50	left
_ - _	50	left
_ * _	40	left
_ && _	40	left
_ / _	40	left
- _	35	right
/ _	35	right
_ ^ _	30	right

Figure 3.1: Notations in the initial state

form	::=	`True`	(`True`)
	|	`False`	(`False`)
	|	~ *form*	(`not`)
	|	*form* /\ *form*	(`and`)
	|	*form* \/ *form*	(`or`)
	|	*form* -> *form*	(*primitive implication*)
	|	*form* <-> *form*	(`iff`)
	|	`forall` *ident* `:` *type* `,` *form*	(*primitive for all*)
	|	`exists` *ident* [`:` *specif*] `,` *form*	(`ex`)
	|	`exists2` *ident* [`:` *specif*] `,` *form* `&` *form*	(`ex2`)
	|	*term* = *term*	(`eq`)
	|	*term* = *term* `:>` *specif*	(`eq`)

Figure 3.2: Syntax of formulas

proposition). The primitive universal quantification allows both first-order and higher-order quantification.

Propositional Connectives

First, we find propositional calculus connectives:

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

Coq < Inductive False :  Prop := .

Coq < Definition not (A: Prop) := A -> False.

Coq < Inductive and (A B:Prop) : Prop := conj (_: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)
Coq <   | or_intror (_:B).

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

Coq < Definition IF_then_else (P Q R:Prop) := P /\ Q \/ ~ P /\ R.
```

Quantifiers

Then we find first-order quantifiers:

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

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

Coq < Inductive ex2 (A:Set) (P Q:A -> Prop) : Prop :=
Coq <     ex_intro2 (x:A) (_:P x) (_:Q x).
```

The following abbreviations are allowed:

`exists x:A, P`	`ex A (fun x:A => P)`
`exists x, P`	`ex _ (fun x => P)`
`exists2 x:A, P & Q`	`ex2 A (fun x:A => P) (fun x:A => Q)`
`exists2 x, P & Q`	`ex2 _ (fun x => P) (fun x => Q)`

The type annotation "`:A`" can be omitted when `A` can be synthesized by the system.

Equality

Then, we find equality, defined as an inductive relation. That is, given a type `A` and an `x` of type `A`, the predicate `(eq A x)` is the smallest one 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:Type) (x:A) : A -> Prop :=
Coq <     refl_equal : eq A x x.
```

Lemmas

Finally, a few easy lemmas are provided.

```
Coq < Theorem absurd : forall A C:Prop, A -> ~ A -> C.

Coq < Section equality.
Coq < Variables A B : Type.
Coq < Variable f : A -> B.
Coq < Variables x y z : A.
Coq < Theorem sym_eq : x = y -> y = x.
Coq < Theorem trans_eq : x = y -> y = z -> x = z.
Coq < Theorem f_equal : x = y -> f x = f y.
Coq < Theorem sym_not_eq : x <> y -> y <> x.

Coq < End equality.
Coq < Definition eq_ind_r :
Coq <   forall (A:Type) (x:A) (P:A -> Prop), P x -> forall y:A, y = x -> P y.
Coq < Definition eq_rec_r :
Coq <   forall (A:Type) (x:A) (P:A -> Set), P x -> forall y:A, y = x -> P y.
Coq < Definition eq_rect_r :
Coq <   forall (A:Type) (x:A) (P:A -> Type), P x -> forall y:A, y = x -> P y.

Coq < Hint Immediate sym_eq sym_not_eq : core.
```

The theorem `f_equal` is extended to functions with two to five arguments. The theorem are names `f_equal2`, `f_equal3`, `f_equal4` and `f_equal5`. For instance `f_equal3` is defined the following way.

```
Coq < Theorem f_equal3 :
Coq <  forall (A1 A2 A3 B:Type) (f:A1 -> A2 -> A3 -> B) (x1 y1:A1) (x2 y2:A2)
Coq <    (x3 y3:A3), x1 = y1 -> x2 = y2 -> x3 = y3 -> f x1 x2 x3 = f y1 y2 y3.
```

3.1.3 Datatypes

In the basic library, we find the definition[2] of the basic data-types of programming, again defined as inductive constructions over the sort `Set`. Some of them come with a special syntax shown on Figure 3.3.

[2] They are in `Datatypes.v`

specif	::=	*specif* * *specif*	(prod)
	|	*specif* + *specif*	(sum)
	|	*specif* + { *specif* }	(sumor)
	|	{ *specif* } + { *specif* }	(sumbool)
	|	{ *ident* : *specif* | *form* }	(sig)
	|	{ *ident* : *specif* | *form* & *form* }	(sig2)
	|	{ *ident* : *specif* & *specif* }	(sigT)
	|	{ *ident* : *specif* & *specif* & *specif* }	(sigT2)
term	::=	(*term* , *term*)	(pair)

Figure 3.3: Syntax of data-types and specifications

Programming

```
Coq < Inductive unit : Set := tt.
Coq < Inductive bool : Set := true | false.
Coq < Inductive nat : Set := O | S (n:nat).
Coq < Inductive option (A:Set) : Set := Some (_:A) | None.
Coq < Inductive identity (A:Type) (a:A) : A -> Type :=
Coq <     refl_identity : identity A a a.
```

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

The predicate `identity` is logically equivalent to equality but it lives in sort `Type`. It is mainly maintained for compatibility.

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 := inl (_:A) | inr (_:B).
Coq < Inductive prod (A B:Set) : Set := pair (_:A) (_:B).
Coq < Section projections.
Coq < Variables A B : Set.
Coq < Definition fst (H: prod A B) := match H with
Coq <                                 | pair x y => x
Coq <                                 end.
Coq < Definition snd (H: prod A B) := match H with
Coq <                                 | pair x y => y
Coq <                                 end.
Coq < End projections.
```

Some operations on `bool` are also provided: `andb` (with infix notation `&&`), `orb` (with infix notation `||`), `xorb`, `implb` and `negb`.

3.1.4 Specification

The following notions[3] allow to build new data-types and specifications. They are available with the syntax shown on Figure 3.3.

[3] They are defined in module `Specif.v`

For instance, given `A:Type` and `P:A->Prop`, the construct `{x:A | P x}` (in abstract syntax `(sig A P)`) is a `Type`. 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 `match`) 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 := exist (x:A) (_:P x).
Coq < Inductive sig2 (A:Set) (P Q:A -> Prop) : Set :=
Coq <   exist2 (x:A) (_:P x) (_:Q x).
```

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

```
Coq < Inductive sigT (A:Type) (P:A -> Type) : Type := existT (x:A) (_:P x).
Coq < Section Projections.
Coq < Variable A : Type.
Coq < Variable P : A -> Type.
Coq < Definition projT1 (H:sigT A P) := let (x, h) := H in x.
Coq < Definition projT2 (H:sigT A P) :=
Coq <   match H return P (projT1 H) with
Coq <     existT x h => h
Coq <   end.
Coq < End Projections.
Coq < Inductive sigT2 (A: Type) (P Q:A -> Type) : Type :=
Coq <     existT2 (x:A) (_:P x) (_:Q x).
```

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

```
Coq < Inductive sumbool (A B:Prop) : Set := left (_:A) | right (_:B).
```

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 := inleft (_:A) | inright (_:B).
```

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

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

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

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

```
Coq < Definition Exc := option.
Coq < Definition value := Some.
Coq < Definition error := None.
```

This module ends with theorems, relating the sorts `Set` or `Type` and `Prop` in a way which is consistent with the realizability interpretation.

```
Coq < Definition except := False_rec.
Coq < Theorem absurd_set : forall (A:Prop) (C:Set), A -> ~ A -> C.
Coq < Theorem and_rect :
Coq <  forall (A B:Prop) (P:Type), (A -> B -> P) -> A /\ B -> P.
```

3.1.5 Basic Arithmetics

The basic library includes a few elementary properties of natural numbers, together with the definitions of predecessor, addition and multiplication[4]. It also provides a scope `nat_scope` gathering standard notations for common operations (+, *) and a decimal notation for numbers. That is he can write `3` for `(S (S (S O)))`. This also works on the left hand side of a `match` expression (see for example section 10.1). This scope is opened by default.

The following example is not part of the standard library, but it shows the usage of the notations:

```
Coq < Fixpoint even (n:nat) : bool :=
Coq <   match n with
Coq <   | 0 => true
Coq <   | 1 => false
Coq <   | S (S n) => even n
Coq <   end.

Coq < Theorem eq_S : forall x y:nat, x = y -> S x = S y.
Coq < Definition pred (n:nat) : nat :=
Coq <   match n with
Coq <   | 0 => 0
Coq <   | S u => u
Coq <   end.
Coq < Theorem pred_Sn : forall m:nat, m = pred (S m).
Coq < Theorem eq_add_S : forall n m:nat, S n = S m -> n = m.
Coq < Hint Immediate eq_add_S : core.
Coq < Theorem not_eq_S : forall n m:nat, n <> m -> S n <> S m.
```

[4] This is in module `Peano.v`

```
Coq < Definition IsSucc (n:nat) : Prop :=
Coq <   match n with
Coq <   | 0 => False
Coq <   | S p => True
Coq <   end.
Coq < Theorem O_S : forall n:nat, 0 <> S n.
Coq < Theorem n_Sn : forall n:nat, n <> S n.

Coq < Fixpoint plus (n m:nat) {struct n} : nat :=
Coq <   match n with
Coq <   | 0 => m
Coq <   | S p => S (p + m)
Coq <   end.
Coq < where "n + m" := (plus n m) : nat_scope.
Coq < Lemma plus_n_O : forall n:nat, n = n + 0.
Coq < Lemma plus_n_Sm : forall n m:nat, S (n + m) = n + S m.

Coq < Fixpoint mult (n m:nat) {struct n} : nat :=
Coq <   match n with
Coq <   | 0 => 0
Coq <   | S p => m + p * m
Coq <   end.
Coq < where "n * m" := (mult n m) : nat_scope.
Coq < Lemma mult_n_O : forall n:nat, 0 = n * 0.
Coq < Lemma mult_n_Sm : forall n m:nat, n * m + n = n * (S m).
```

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 : forall m:nat, n <= m -> n <= (S m).
Coq < where "n <= m" := (le n m) : nat_scope.
Coq < Definition lt (n m:nat) := S n <= m.
Coq < Definition ge (n m:nat) := m <= n.
Coq < Definition gt (n m:nat) := m < n.
```

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 :
Coq <  forall (n:nat) (P:nat -> Prop), P 0 -> (forall m:nat, P (S m)) -> P n.

Coq < Theorem nat_double_ind :
Coq <  forall R:nat -> nat -> Prop,
Coq <    (forall n:nat, R 0 n) ->
Coq <    (forall n:nat, R (S n) 0) ->
Coq <    (forall n m:nat, R n m -> R (S n) (S m)) -> forall n m:nat, R n m.
```

3.1.6 Well-founded recursion

The basic library contains the basics of well-founded recursion and well-founded induction[5].

```
Coq < Section Well_founded.

Coq < Variable A : Type.

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

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

Coq < Lemma Acc_inv : Acc x -> forall y:A, R y x -> Acc y.


Coq < Definition well_founded := forall a:A, Acc a.

Coq < Hypothesis Rwf : well_founded.

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

Coq < Theorem well_founded_ind :
Coq <  forall P:A -> Prop,
Coq <    (forall x:A, (forall y:A, R y x -> P y) -> P x) -> forall a:A, P a.
```

The automatically generated scheme `Acc_rect` can be used to define functions by fixpoints using well-founded relations to justify termination. Assuming extensionality of the functional used for the recursive call, the fixpoint equation can be proved.

```
Coq < Section FixPoint.

Coq < Variable P : A -> Type.

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

Coq < Fixpoint Fix_F (x:A) (r:Acc x) {struct r} : P x :=
Coq <   F x (fun (y:A) (p:R y x) => Fix_F y (Acc_inv x r y p)).

Coq < Definition Fix (x:A) := Fix_F x (Rwf x).

Coq < Hypothesis F_ext :
Coq <     forall (x:A) (f g:forall y:A, R y x -> P y),
Coq <       (forall (y:A) (p:R y x), f y p = g y p) -> F x f = F x g.

Coq < Lemma Fix_F_eq :
Coq <  forall (x:A) (r:Acc x),
Coq <    F x (fun (y:A) (p:R y x) => Fix_F y (Acc_inv x r y p)) = Fix_F x r.

Coq < Lemma Fix_F_inv : forall (x:A) (r s:Acc x), Fix_F x r = Fix_F x s.

Coq < Lemma fix_eq : forall x:A, Fix x = F x (fun (y:A) (p:R y x) => Fix y).


Coq < End FixPoint.

Coq < End Well_founded.
```

[5] This is defined in module `Wf.v`

3.1.7 Accessing the Type level

The basic library includes the definitions[6] of the counterparts of some data-types and logical quantifiers at the `Type` level: negation, pair, and properties of `identity`.

```
Coq < Definition notT (A:Type) := A -> False.
Coq < Inductive prodT (A B:Type) : Type := pairT (_:A) (_:B).
```

At the end, it defines data-types at the Type level.

3.1.8 Tactics

A few tactics defined at the user level are provided in the initial state[7].

3.2 The standard library

3.2.1 Survey

The rest of the standard library is structured into the following subdirectories:

Logic	Classical logic and dependent equality
Arith	Basic Peano arithmetic
NArith	Basic positive integer arithmetic
ZArith	Basic relative integer arithmetic
Numbers	Various approaches to natural, integer and cyclic numbers (currently axiomatically and on top of 2^{31} binary words)
Bool	Booleans (basic functions and results)
Lists	Monomorphic and polymorphic lists (basic functions and results), Streams (infinite sequences defined with co-inductive types)
Sets	Sets (classical, constructive, finite, infinite, power set, etc.)
FSets	Specification and implementations of finite sets and finite maps (by lists and by AVL trees)
Reals	Axiomatization of real numbers (classical, basic functions, integer part, fractional part, limit, derivative, Cauchy series, power series and results,...)
Relations	Relations (definitions and basic results)
Sorting	Sorted list (basic definitions and heapsort correctness)
Strings	8-bits characters and strings
Wellfounded	Well-founded relations (basic results)

These directories belong to the initial load path of the system, and the modules they provide are compiled at installation time. So they are directly accessible with the command `Require` (see Chapter 6).

The different modules of the COQ standard library are described in the additional document `Library.dvi`. They are also accessible on the WWW through the COQ homepage [8].

[6] This is in module `Logic_Type.v`
[7] This is in module `Tactics.v`
[8] `http://coq.inria.fr`

Notation	Interpretation	Precedence	Associativity
_ < _	Zlt		
x <= y	Zle		
_ > _	Zgt		
x >= y	Zge		
x < y < z	x < y /\ y < z		
x < y <= z	x < y /\ y <= z		
x <= y < z	x <= y /\ y < z		
x <= y <= z	x <= y /\ y <= z		
_ ?= _	Zcompare	70	no
_ + _	Zplus		
_ - _	Zminus		
_ * _	Zmult		
_ / _	Zdiv		
_ mod _	Zmod	40	no
- _	Zopp		
_ ^ _	Zpower		

Figure 3.4: Definition of the scope for integer arithmetics (Z_scope)

3.2.2 Notations for integer arithmetics

On Figure 3.2.2 is described the syntax of expressions for integer arithmetics. It is provided by requiring and opening the module ZArith and opening scope Z_scope.

Figure 3.2.2 shows the notations provided by Z_scope. It specifies how notations are interpreted and, when not already reserved, the precedence and associativity.

```
Coq < Require Import ZArith.

Coq < Check  (2 + 3)%Z.
(2 + 3)%Z
     : Z

Coq < Open Scope Z_scope.

Coq < Check 2 + 3.
2 + 3
     : Z
```

3.2.3 Peano's arithmetic (nat)

While in the initial state, many operations and predicates of Peano's arithmetic are defined, further operations and results belong to other modules. For instance, the decidability of the basic predicates are defined here. This is provided by requiring the module Arith.

Figure 3.2.3 describes notation available in scope nat_scope.

Notation	Interpretation
`_ < _`	`lt`
`x <= y`	`le`
`_ > _`	`gt`
`x >= y`	`ge`
`x < y < z`	`x < y /\ y < z`
`x < y <= z`	`x < y /\ y <= z`
`x <= y < z`	`x <= y /\ y < z`
`x <= y <= z`	`x <= y /\ y <= z`
`_ + _`	`plus`
`_ - _`	`minus`
`_ * _`	`mult`

Figure 3.5: Definition of the scope for natural numbers (`nat_scope`)

Notation	Interpretation
`_ < _`	`Rlt`
`x <= y`	`Rle`
`_ > _`	`Rgt`
`x >= y`	`Rge`
`x < y < z`	`x < y /\ y < z`
`x < y <= z`	`x < y /\ y <= z`
`x <= y < z`	`x <= y /\ y < z`
`x <= y <= z`	`x <= y /\ y <= z`
`_ + _`	`Rplus`
`_ - _`	`Rminus`
`_ * _`	`Rmult`
`_ / _`	`Rdiv`
`- _`	`Ropp`
`/ _`	`Rinv`
`_ ^ _`	`pow`

Figure 3.6: Definition of the scope for real arithmetics (`R_scope`)

3.2.4 Real numbers library

Notations for real numbers

This is provided by requiring and opening the module `Reals` and opening scope `R_scope`. This set of notations is very similar to the notation for integer arithmetics. The inverse function was added.

```
Coq < Require Import Reals.
Coq < Check  (2 + 3)%R.
(2 + 3)%R
     : R
Coq < Open Scope R_scope.
Coq < Check 2 + 3.
```

```
2 + 3
     : R
```

Some tactics

In addition to the `ring`, `field` and `fourier` tactics (see Chapter 8) there are:

- `discrR`

  Proves that a real integer constant c_1 is different from another real integer constant c_2.

```
Coq < Require Import DiscrR.
Coq < Goal 5 <> 0.

Coq < discrR.
Proof completed.
```

- `split_Rabs` allows to unfold `Rabs` constant and splits corresponding conjonctions.

```
Coq < Require Import SplitAbsolu.
Coq < Goal forall x:R, x <= Rabs x.

Coq < intro; split_Rabs.
2 subgoals

  x : R
  r : x < 0
  ============================
   x <= - x
subgoal 2 is:
 x <= x
```

- `split_Rmult` allows to split a condition that a product is non null into subgoals corresponding to the condition on each operand of the product.

```
Coq < Require Import SplitRmult.
Coq < Goal forall x y z:R, x * y * z <> 0.

Coq < intros; split_Rmult.
3 subgoals

  x : R
  y : R
  z : R
  ============================
   x <> 0
subgoal 2 is:
 y <> 0
subgoal 3 is:
 z <> 0
```

All this tactics has been written with the tactic language Ltac described in Chapter 9. More details are available in document http://coq.inria.fr/~desmettr/Reals.ps.

Notation	Interpretation	Precedence	Associativity
_ ++ _	app	60	right
_ :: _	cons	60	right

Figure 3.7: Definition of the scope for lists (list_scope)

3.2.5 List library

Some elementary operations on polymorphic lists are defined here. They can be accessed by requiring module List.

It defines the following notions:

length	length
head	first element (with default)
tail	all but first element
app	concatenation
rev	reverse
nth	accessing n-th element (with default)
map	applying a function
flat_map	applying a function returning lists
fold_left	iterator (from head to tail)
fold_right	iterator (from tail to head)

Table show notations available when opening scope list_scope.

3.3 Users' contributions

Numerous users' contributions have been collected and are available at URL coq.inria.fr/contribs/. On this web page, you have a list of all contributions with informations (author, institution, quick description, etc.) and the possibility to download them one by one. There is a small search engine to look for keywords in all contributions. You will also find informations on how to submit a new contribution.

The users' contributions may also be obtained by anonymous FTP from site ftp.inria.fr, in directory INRIA/coq/ and searchable on-line at http://coq.inria.fr/contribs-eng.html

Chapter 4

Calculus of Inductive Constructions

The underlying formal language of Coq is a *Calculus of Constructions* with *Inductive Definitions*. It is presented in this chapter. For Coq version V7, this Calculus was known as the *Calculus of (Co)Inductive Constructions* (Cic in short). The underlying calculus of Coq version V8.0 and up is a weaker calculus where the sort Set satisfies predicative rules. We call this calculus the *Predicative Calculus of (Co)Inductive Constructions* (pCic in short). In Section 4.7 we give the extra-rules for Cic. A compiling option of Coq allows to type-check theories in this extended system.

In pCic all objects have a *type*. There are types for functions (or programs), there are atomic types (especially datatypes)... but also types for proofs and types for the types themselves. Especially, any object handled in the formalism must 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 has type T"*. The terms of pCic are detailed in Section 4.1.

In pCic there is an internal reduction mechanism. In particular, it allows to decide if two programs are *intentionally* equal (one says *convertible*). Convertibility is presented in section 4.3.

The remaining sections are concerned with the type-checking of terms. The beginner can skip them.

The reader seeking a background on the Calculus of Inductive Constructions may read several papers. Giménez and Castéran [68] provide an introduction to inductive and coinductive definitions in Coq. In their book [14], Bertot and Castéran give a precise description of the pCic based on numerous practical examples. Barras [9], Werner [135] and Paulin-Mohring [117] are the most recent theses dealing with Inductive Definitions. Coquand-Huet [29, 30, 31] introduces the Calculus of Constructions. Coquand-Paulin [32] extended this calculus to inductive definitions. The pCic is a formulation of type theory including the possibility of inductive constructions, Barendregt [6] studies the modern form of type theory.

4.1 The terms

In most type theories, one usually makes a syntactic distinction between types and terms. This is not the case for pCic 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.

Consider for instance the $\rightarrow$ constructor and assume nat is the type of natural numbers. Then $\rightarrow$ is used both to denote nat $\rightarrow$ nat which is the type of functions from nat to nat, and to denote nat $\rightarrow$ Prop which is the type of unary predicates over the natural numbers. Consider abstraction which

builds functions. It serves to build "ordinary" functions as fun x : nat $\Rightarrow$ (`mult` x x) (assuming `mult` is already defined) but may build also predicates over the natural numbers. For instance fun x : nat $\Rightarrow$ $(x = x)$ will represent a predicate P, informally written in mathematics $P(x) \equiv x = x$. If P has type nat $\rightarrow$ Prop, $(P\ x)$ is a proposition, furthermore forall x : 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 "$\forall x.P(x)$".

4.1.1 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 pCIC 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 called a *proposition*.

The sort Set intends to be the type of specifications. This includes programs and the usual sets such as booleans, naturals, lists etc.

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 pCIC. These are, in addition to Set and Prop a hierarchy of universes Type(i) for any integer i. We call $\mathcal{S}$ the set of sorts which is defined by:

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

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

The user will never mention explicitly the index i when referring to the universe Type(i). One only writes Type. The system itself generates for each instance of Type 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.

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

Implementation issues In practice, the Type hierarchy is implemented using algebraic universes. An algebraic universe u is either a variable (a qualified identifier with a number) or a successor of an algebraic universe (an expression $u + 1$), or an upper bound of algebraic universes (an expression $max(u_1, ..., u_n)$), or the base universe (the expression 0) which corresponds, in the arity of sort-polymorphic inductive types, to the predicative sort Set. A graph of constraints between the universe variables is maintained globally. To ensure the existence of a mapping of the universes to the positive integers, the graph of constraints must remain acyclic. Typing expressions that violate the acyclicity of the graph of constraints results in a `Universe inconsistency` error (see also Section 2.10).

4.1.2 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 pCIC, 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.

4.1.3 Terms

Terms are built from variables, global names, constructors, abstraction, application, local declarations bindings ("let-in" expressions) and product.

From a syntactic point of view, types cannot be distinguished from terms, except that they cannot start by an abstraction, and that if a term is a sort or a product, it should be a type.

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

1. the sorts Set, Prop, Type are terms.
2. names for global constants of the environment are terms.
3. variables are terms.
4. if x is a variable and T, U are terms then $\forall\ x : T, U$ (forall $x : T, U$ in Coq concrete syntax) is a term. If x occurs in U, $\forall\ x : T, U$ reads as *"for all x of type T, U"*. As U depends on x, one says that $\forall\ x : T, U$ is a *dependent product*. If x doesn't occurs in U then $\forall\ x : T, U$ 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 $\lambda\ x : T, U$ (fun $x : T \Rightarrow U$ in Coq concrete syntax) is a term. This is a notation for the λ-abstraction of λ-calculus [8]. The term $\lambda\ x : T, U$ is a function which maps elements of T to U.
6. if T and U are terms then $(T\ U)$ is a term ($T\ U$ in Coq concrete syntax). The term $(T\ U)$ reads as *"T applied to U"*.
7. if x is a variable, and T, U are terms then let $x := T$ in U is a term which denotes the term U where the variable x is locally bound to T. This stands for the common "let-in" construction of functional programs such as ML or Scheme.

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

Free variables. The notion of free variables is defined as usual. In the expressions $\lambda\ x : T, U$ and $\forall x : T, U$ 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 is written $u\{x/t\}$.

4.2 Typed terms

As objects of type theory, terms are subjected to *type discipline*. The well typing of a term depends on an environment which consists in a global environment (see below) and a local context.

Local context. A *local context* (or shortly context) is an ordered list of declarations of variables. The declaration of some variable x is either an assumption, written $x : T$ (T is a type) or a definition, written $x := t : T$. We use brackets to write contexts. A typical example is $[x : T; y := u : U; z : V]$. Notice that the variables declared in a context must be distinct. If Γ declares some x, we write $x \in \Gamma$. By writing $(x : T) \in \Gamma$ we mean that either $x : T$ is an assumption in Γ or that there exists some t such that $x := t : T$ is a definition in Γ. If Γ defines some $x := t : T$, we also write $(x := t : T) \in \Gamma$. Contexts must be themselves *well formed*. For the rest of the chapter, the notation $\Gamma :: (y : T)$ (resp. $\Gamma :: (y := t : T)$) denotes the context Γ enriched with the declaration $y : T$ (resp. $y := t : T$). 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, type T and term t, if $(x : T) \in \Gamma$ then $(x : T) \in \Delta$ and if $(x := t : T) \in \Gamma$ then $(x := t : T) \in \Delta$.

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 global declarations (constants and global assumptions), we also need to consider a global environment E.

An environment is an ordered list of declarations of global names. Declarations are either assumptions or "standard" definitions, that is abbreviations for well-formed terms but also definitions of inductive objects. In the latter case, an object in the environment will define one or more constants (that is types and constructors, see Section 4.5).

An assumption will be represented in the environment as $\mathsf{Assum}(\Gamma)(c : T)$ which means that c is assumed of some type T well-defined in some context Γ. An (ordinary) 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 some context Γ whose value is t and type is T.

The rules for inductive definitions (see section 4.5) have to be considered as assumption rules to which the following definitions apply: if the name c is declared in E, we write $c \in E$ and if $c : T$ or $c := t : T$ is declared in E, we write $(c : T) \in E$.

Typing rules. In the following, we assume E is a valid environment wrt to inductive definitions. 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 [1].

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.

[1] 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 subcontext of the current context.

W-E

$$\mathcal{WF}([])[[]]$$

W-S

$$\frac{E[\Gamma] \vdash T : s \quad s \in \mathcal{S} \quad x \notin \Gamma}{\mathcal{WF}(E)[\Gamma :: (x : T)]} \qquad \frac{E[\Gamma] \vdash t : T \quad x \notin \Gamma}{\mathcal{WF}(E)[\Gamma :: (x := t : T)]}$$

Def

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

Assum

$$\frac{E[\Gamma] \vdash T : s \quad s \in \mathcal{S} \quad c \notin E \cup \Gamma}{\mathcal{WF}(E; \mathsf{Assum}(\Gamma)(c : T))[\Gamma]}$$

Ax

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

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

Var

$$\frac{\mathcal{WF}(E)[\Gamma] \quad (x : T) \in \Gamma \text{ or } (x := t : T) \in \Gamma \text{ for some } t}{E[\Gamma] \vdash x : T}$$

Const

$$\frac{\mathcal{WF}(E)[\Gamma] \quad (c : T) \in E \text{ or } (c := t : T) \in E \text{ for some } t}{E[\Gamma] \vdash c : T}$$

Prod

$$\frac{E[\Gamma] \vdash T : s \quad s \in \mathcal{S} \quad E[\Gamma :: (x : T)] \vdash U : \mathsf{Prop}}{E[\Gamma] \vdash \forall\, x : T, U : \mathsf{Prop}}$$

$$\frac{E[\Gamma] \vdash T : s \quad s \in \{\mathsf{Prop}, \mathsf{Set}\} \quad E[\Gamma :: (x : T)] \vdash U : \mathsf{Set}}{E[\Gamma] \vdash \forall\, x : T, U : \mathsf{Set}}$$

$$\frac{E[\Gamma] \vdash T : \mathsf{Type}(i) \quad i \leq k \quad E[\Gamma :: (x : T)] \vdash U : \mathsf{Type}(j) \quad j \leq k}{E[\Gamma] \vdash \forall\, x : T, U : \mathsf{Type}(k)}$$

Lam

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

App

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

Let

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

Remark: We may have $\mathsf{let}\ x := t\ \mathsf{in}\ u$ well-typed without having $((\lambda\ x : T, u)\ t)$ well-typed (where T is a type of t). This is because the value t associated to x may be used in a conversion rule (see Section 4.3).

4.3 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 $\lambda\, x : T, x$. In any environment E and context Γ, we want to identify any object a (of type T) with the application $((\lambda\, x : T, x)\ a)$. We define for this a *reduction* (or a *conversion*) rule we call β:

$$E[\Gamma] \vdash ((\lambda\, x : T, t)\ u) \triangleright_\beta t\{x/u\}$$

We say that $t\{x/u\}$ is the *β-contraction* of $((\lambda\, x : T, t)\ u)$ and, conversely, that $((\lambda\, x : T, t)\ u)$ 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 [23].

ι-reduction. A specific conversion rule is associated to the inductive objects in the environment. We shall give later on (see Section 4.5.4) 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 [116, 135].

δ-reduction. We may have defined variables in contexts or constants in the global environment. It is legal to identify such a reference with its value, that is to expand (or unfold) it into its value. This reduction is called δ-reduction and shows as follows.

$$E[\Gamma] \vdash x \triangleright_\delta t \quad \text{if } (x := t : T) \in \Gamma \qquad E[\Gamma] \vdash c \triangleright_\delta t \quad \text{if } (c := t : T) \in E$$

ζ-reduction. Coq allows also to remove local definitions occurring in terms by replacing the defined variable by its value. The declaration being destroyed, this reduction differs from δ-reduction. It is called ζ-reduction and shows as follows.

$$E[\Gamma] \vdash \mathsf{let}\ x := u\ \mathsf{in}\ t \triangleright_\zeta t\{x/u\}$$

Convertibility. Let us write $E[\Gamma] \vdash t \triangleright u$ for the contextual closure of the relation t reduces to u in the environment E and context Γ with one of the previous reduction β, ι, δ or ζ.

We say that two terms t_1 and t_2 are *convertible* (or *equivalent)* in the environment E and context Γ iff there exists a term u such that $E[\Gamma] \vdash t_1 \triangleright \ldots \triangleright u$ and $E[\Gamma] \vdash t_2 \triangleright \ldots \triangleright u$. We then write $E[\Gamma] \vdash t_1 =_{\beta\delta\iota\zeta} t_2$.

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 $E[\Gamma] \vdash t =_{\beta\delta\iota\zeta} u$ then $E[\Gamma] \vdash t \leq_{\beta\delta\iota\zeta} u$,
2. if $i \leq j$ then $E[\Gamma] \vdash \mathsf{Type}(i) \leq_{\beta\delta\iota\zeta} \mathsf{Type}(j)$,

3. for any i, $E[\Gamma] \vdash \mathsf{Prop} \leq_{\beta\delta\iota\zeta} \mathsf{Type}(i)$,
4. for any i, $E[\Gamma] \vdash \mathsf{Set} \leq_{\beta\delta\iota\zeta} \mathsf{Type}(i)$,
5. if $E[\Gamma] \vdash T =_{\beta\delta\iota\zeta} U$ and $E[\Gamma :: (x : T)] \vdash T' \leq_{\beta\delta\iota\zeta} U'$ then $E[\Gamma] \vdash \forall\, x : T, T' \leq_{\beta\delta\iota\zeta} \forall\, x : U, U'$.

The conversion rule is now exactly:

Conv

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

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

$$E[\Gamma] \vdash \lambda\, x : T, (t\ x) \triangleright t$$

Indeed, as x doesn't occur free in t, for any u one applies to $\lambda\, x : T, (t\ x)$, it β-reduces to $(t\ u)$. So $\lambda\, x : T, (t\ x)$ 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 8). Any term can be written as $\lambda\, x_1 : T_1, \ldots \lambda x_k : T_k, (t_0\ t_1 \ldots t_n)$ 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 $\lambda\, x : T, u_0$ then one step of β-head reduction of t is:

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

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:

$$t \triangleright \ldots \triangleright \lambda\, x_1 : T_1, \ldots \lambda x_k : T_k, (v\ u_1 \ldots u_m)$$

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, ι and ζ reductions or any combination of those can also be defined.

4.4 Derived rules for environments

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:

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

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:

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

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:

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

4.5 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 states that a list has some given length and the mutual inductive definition of trees and forests.

4.5.1 Representing an inductive definition

Inductive definitions without parameters

As for constants, inductive definitions can be defined in a non-empty context.
We write $\mathsf{Ind}(\Gamma)(\Gamma_I := \Gamma_C)$ an inductive definition valid in a context Γ, a context of definitions Γ_I and a context of constructors Γ_C.

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

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

In a context with a variable $A : \mathsf{Set}$, the lists of elements in A are represented by:

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

Assuming Γ_I is $[I_1 : A_1; \ldots; I_k : A_k]$, and Γ_C is $[c_1 : C_1; \ldots; c_n : C_n]$, the general typing rules are, for $1 \leq j \leq k$ and $1 \leq i \leq n$:

$$\frac{\mathsf{Ind}(\Gamma)(\Gamma_I := \Gamma_C) \ \in E}{(I_j : A_j) \in E}$$

$$\frac{\mathsf{Ind}(\Gamma)(\Gamma_I := \Gamma_C) \ \in E}{(c_i : C_i) \in E}$$

Inductive definitions with parameters

We have to slightly complicate the representation above in order to handle the delicate problem of parameters. Let us explain that on the example of List. With the above definition, the type List can only be used in an environment where we have a variable $A : \mathsf{Set}$. Generally one want to consider lists of elements in different types. For constants this is easily done by abstracting the value over the parameter. In the case of inductive definitions we have to handle the abstraction over several objects.

One possible way to do that would be to define the type List inductively as being an inductive family of type $\mathsf{Set} \to \mathsf{Set}$:

$$\mathsf{Ind}()(\mathsf{List} : \mathsf{Set} \to \mathsf{Set} := \mathsf{nil} : (\forall A : \mathsf{Set}, \mathsf{List}\ A), \mathsf{cons} : (\forall A : \mathsf{Set}, A \to \mathsf{List}\ A \to \mathsf{List}\ A))$$

There are drawbacks to this point of view. The information which says that for any A, (List A) is an inductively defined Set has been lost. So we introduce two important definitions.

Inductive parameters, real arguments. An inductive definition $\mathsf{Ind}(\Gamma)(\Gamma_I := \Gamma_C)$ admits r inductive parameters if each type of constructors $(c : C)$ in Γ_C is such that

$$C \equiv \forall p_1 : P_1, \ldots, \forall p_r : P_r, \forall a_1 : A_1, \ldots \forall a_n : A_n, (I\ p_1\ \ldots p_r\ t_1 \ldots t_q)$$

with I one of the inductive definitions in Γ_I. We say that q is the number of real arguments of the constructor c.

Context of parameters. If an inductive definition $\mathsf{Ind}(\Gamma)(\Gamma_I := \Gamma_C)$ admits r inductive parameters, then there exists a context Γ_P of size r, such that $\Gamma_P = [p_1 : P_1; \ldots; p_r : P_r]$ and if $(t : A) \in \Gamma_I, \Gamma_C$ then A can be written as $\forall p_1 : P_1, \ldots \forall p_r : P_r, A'$. We call Γ_P the context of parameters of the inductive definition and use the notation $\forall \Gamma_P, A'$ for the term A.

Remark. If we have a term t in an instance of an inductive definition I which starts with a constructor c, then the r first arguments of c (the parameters) can be deduced from the type T of t: these are exactly the r first arguments of I in the head normal form of T.

Examples. The List definition has 1 parameter:

$$\mathsf{Ind}()(\mathsf{List} : \mathsf{Set} \to \mathsf{Set} := \mathsf{nil} : (\forall A : \mathsf{Set}, \mathsf{List}\ A), \mathsf{cons} : (\forall A : \mathsf{Set}, A \to \mathsf{List}\ A \to \mathsf{List}\ A)\ )$$

This is also the case for this more complex definition where there is a recursive argument on a different instance of List:

$$\mathsf{Ind}()(\mathsf{List} : \mathsf{Set} \to \mathsf{Set} := \mathsf{nil} : (\forall A : \mathsf{Set}, \mathsf{List}\ A), \mathsf{cons} : (\forall A : \mathsf{Set}, A \to \mathsf{List}\ (A \to A) \to \mathsf{List}\ A)\ )$$

But the following definition has 0 parameters:

$$\mathsf{Ind}()(\mathsf{List} : \mathsf{Set} \to \mathsf{Set} := \mathsf{nil} : (\forall A : \mathsf{Set}, \mathsf{List}\ A), \mathsf{cons} : (\forall A : \mathsf{Set}, A \to \mathsf{List}\ A \to \mathsf{List}\ (A * A))\ )$$

Concrete syntax. In the Coq system, the context of parameters is given explicitly after the name of the inductive definitions and is shared between the arities and the type of constructors. We keep track in the syntax of the number of parameters.

Formally the representation of an inductive declaration will be $\mathsf{Ind}(\Gamma)[p](\Gamma_I := \Gamma_C\ )$ for an inductive definition valid in a context Γ with p parameters, a context of definitions Γ_I and a context of constructors Γ_C.

The definition $\mathsf{Ind}(\Gamma)[p](\Gamma_I := \Gamma_C\ )$ will be well-formed exactly when $\mathsf{Ind}(\Gamma)(\Gamma_I := \Gamma_C\ )$ is and when p is (less or equal than) the number of parameters in $\mathsf{Ind}(\Gamma)(\Gamma_I := \Gamma_C\ )$.

Examples The declaration for parameterized lists is:

$$\mathsf{Ind}()[1](\mathsf{List} : \mathsf{Set} \to \mathsf{Set} := \mathsf{nil} : (\forall A : \mathsf{Set}, \mathsf{List}\ A), \mathsf{cons} : (\forall A : \mathsf{Set}, A \to \mathsf{List}\ A \to \mathsf{List}\ A)\ )$$

The declaration for the length of lists is:

$$\mathsf{Ind}()[1](\mathsf{Length} : \forall A : \mathsf{Set}, (\mathsf{List}\ A) \to \mathsf{nat} \to \mathsf{Prop} := \mathsf{Lnil} : \forall A : \mathsf{Set}, \mathsf{Length}\ A\ (\mathsf{nil}\ A)\ \mathsf{O},$$
$$\mathsf{Lcons} : \forall A : \mathsf{Set}, \forall a : A, \forall l : (\mathsf{List}\ A), \forall n : \mathsf{nat}, (\mathsf{Length}\ A\ l\ n) \to (\mathsf{Length}\ A\ (\mathsf{cons}\ A\ a\ l)\ (\mathsf{S}\ n))\ )$$

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

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

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

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

The COQ type-checker verifies that all parameters are applied in the correct manner in the conclusion of the type of each constructors :

In particular, the following definition will not be accepted because there is an occurrence of List which is not applied to the parameter variable in the conclusion of the type of `cons'`:

```
Coq < Inductive list' (A:Set) : Set :=
Coq <   | nil' : list' A
Coq <   | cons' : A -> list' A -> list' (A*A).
Coq < Coq < Error: Last occurrence of "list'" must have "A" as 1st argument in
 "A -> list' A -> list' (A * A)%type".
```

Since COQ version 8.1, there is no restriction about parameters in the types of arguments of constructors. The following definition is valid:

```
Coq < Inductive list' (A:Set) : Set :=
Coq <   | nil' : list' A
Coq <   | cons' : A -> list' (A->A) -> list' A.
list' is defined
list'_rect is defined
list'_ind is defined
list'_rec is defined
```

4.5.2 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_I is $[I_1 : A_1; \ldots; I_k : A_k]$, and Γ_C is $[c_1 : C_1; \ldots; c_n : C_n]$,

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

$$\frac{\mathsf{Ind}(\Gamma)[p](\Gamma_I := \Gamma_C) \in E \quad i = 1..n}{(c_i : C_i) \in E}$$

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

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

4.5.3 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 converts to the sort s or to a product $\forall\ x : T, U$ with U an arity of sort s. (For instance $A \to \mathsf{Set}$ or $\forall\ A : \mathsf{Prop}, A \to \mathsf{Prop}$ are arities of sort respectively Set and Prop). A *type of constructor of* I is either a term $(I\ t_1 \ldots\ t_n)$ or $\forall x : T, C$ with C recursively a *type of constructor of* I.

The type of constructor T will be said to *satisfy the positivity condition* for a constant X in the following cases:

- $T = (X\ t_1 \ldots\ t_n)$ and X does not occur free in any t_i
- $T = \forall\ x : U, V$ and X occurs only strictly positively in U and the type V satisfies the positivity condition for X

The constant X *occurs strictly positively* in T in the following cases:

- X does not occur in T
- T converts to $(X\ t_1 \ldots\ t_n)$ and X does not occur in any of t_i
- T converts to $\forall\ x : U, V$ and X does not occur in type U but occurs strictly positively in type V
- T converts to $(I\ a_1 \ldots\ a_m\ t_1 \ldots\ t_p)$ where I is the name of an inductive declaration of the form $\mathsf{Ind}(\Gamma)[m](I : A := c_1 : \forall p_1 : P_1, \ldots \forall p_m : P_m, C_1; \ldots; c_n : \forall p_1 : P_1, \ldots \forall p_m : P_m, C_n\,)$ (in particular, it is not mutually defined and it has m parameters) and X does not occur in any of the t_i, and the (instantiated) types of constructor $C_i\{p_j/a_j\}_{j=1\ldots m}$ of I satisfy the nested positivity condition for X

The type of constructor T of I *satisfies the nested positivity condition* for a constant X in the following cases:

- $T = (I\ b_1 \ldots b_m\ u_1 \ldots\ u_p)$, I is an inductive definition with m parameters and X does not occur in any u_i
- $T = \forall\ x : U, V$ and X occurs only strictly positively in U and the type V satisfies the nested positivity condition for X

Example X occurs strictly positively in $A \to X$ or $X * A$ or $(\texttt{list}\ X)$ but not in $X \to A$ or $(X \to A) \to A$ nor $(\texttt{neg}\ A)$ assuming the notion of product and lists were already defined and `neg` is an inductive definition with declaration $\mathsf{Ind}()[A : \mathsf{Set}](\texttt{neg} : \mathsf{Set} := \texttt{neg} : (A \to \texttt{False}) \to \texttt{neg}\,)$. Assuming X has arity $\texttt{nat} \to \texttt{Prop}$ and `ex` is the inductively defined existential quantifier, the occurrence of X in $(\texttt{ex nat}\ \lambda\ \texttt{n} : \texttt{nat}, (\texttt{X n}))$ is also strictly positive.

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 $[I_1 : \forall \Gamma_P, A_1; \ldots; I_k : \forall \Gamma_P, A_k]$ and Γ_C is $[c_1 : \forall \Gamma_P, C_1; \ldots; c_n : \forall \Gamma_P, C_n]$.

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

provided that the following side conditions hold:

- $k > 0$ and all of I_j and c_i are distinct names for $j = 1 \ldots k$ and $i = 1 \ldots n$,
- p is the number of parameters of $\mathsf{Ind}(\Gamma)(\Gamma_I := \Gamma_C\,)$ and Γ_P is the context of parameters,
- for $j = 1 \ldots k$ we have that A_j is an arity of sort s_j and $I_j \notin \Gamma \cup E$,
- for $i = 1 \ldots n$ we have that C_i is a type of constructor of I_{q_i} which satisfies the positivity condition for $I_1 \ldots I_k$ 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 the impredicative sort (Prop) but may fail to define inductive definition on sort Set and generate constraints between universes for inductive definitions in the Type hierarchy.

Examples. It is well known that existential quantifier can be encoded as an inductive definition. The following declaration introduces the second-order existential quantifier $\exists X.P(X)$.

```
Coq < Inductive exProp (P:Prop->Prop) : Prop
Coq <   := exP_intro : forall X:Prop, P X -> exProp P.
```

The same definition on Set is not allowed and fails :

```
Coq < Inductive exSet (P:Set->Prop) : Set
Coq <   := exS_intro : forall X:Set, P X -> exSet P.
Coq < Coq < Error: Large non-propositional inductive types must be in Type.
```

It is possible to declare the same inductive definition in the universe Type. The `exType` inductive definition has type $(\mathsf{Type}_i \to \mathsf{Prop}) \to \mathsf{Type}_j$ with the constraint that the parameter `X` of `exT_intro` has type Type_k with $k < j$ and $k \leq i$.

```
Coq < Inductive exType (P:Type->Prop) : Type
Coq <   := exT_intro : forall X:Type, P X -> exType P.
```

Sort-polymorphism of inductive families. From Coq version 8.1, inductive families declared in Type are polymorphic over their arguments in Type.

If A is an arity and s a sort, we write $A_{/s}$ for the arity obtained from A by replacing its sort with s. Especially, if A is well-typed in some environment and context, then $A_{/s}$ is typable by typability of all products in the Calculus of Inductive Constructions. The following typing rule is added to the theory.

Ind-Family Let $\mathsf{Ind}(\Gamma)[p](\Gamma_I := \Gamma_C)$ be an inductive definition. Let $\Gamma_P = [p_1 : P_1; \ldots; p_p : P_p]$ be its context of parameters, $\Gamma_I = [I_1 : \forall \Gamma_P, A_1; \ldots; I_k : \forall \Gamma_P, A_k]$ its context of definitions and $\Gamma_C = [c_1 : \forall \Gamma_P, C_1; \ldots; c_n : \forall \Gamma_P, C_n]$ its context of constructors, with c_i a constructor of I_{q_i}.

Let $m \leq p$ be the length of the longest prefix of parameters such that the m first arguments of all occurrences of all I_j in all C_k (even the occurrences in the hypotheses of C_k) are exactly applied to $p_1 \ldots p_m$ (m is the number of *recursively uniform parameters* and the $p-m$ remaining parameters are the *recursively non-uniform parameters*). Let $q_1, \ldots, q_r$, with $0 \leq r \leq m$, be a (possibly) partial instantiation of the recursively uniform parameters of Γ_P. We have:

$$\frac{\left\{\begin{array}{l}\mathsf{Ind}(\Gamma)[p](\Gamma_I := \Gamma_C) \in E\\ (E[\Gamma] \vdash q_l : P'_l)_{l=1\ldots r}\\ (E[\Gamma] \vdash P'_l \leq_{\beta\delta\iota\zeta} P_l\{p_u/q_u\}_{u=1\ldots l-1})_{l=1\ldots r}\\ 1 \leq j \leq k\end{array}\right.}{E[\Gamma] \vdash (I_j\, q_1\, \ldots\, q_r : \forall [p_{r+1} : P_{r+1}; \ldots; p_p : P_p], (A_j)_{/s_j})}$$

provided that the following side conditions hold:

- $\Gamma_{P'}$ is the context obtained from Γ_P by replacing each P_l that is an arity with P'_l for $1 \leq l \leq r$ (notice that P_l arity implies P'_l arity since $E[\Gamma] \vdash P'_l \leq_{\beta\delta\iota\zeta} P_l\{p_u/q_u\}_{u=1\ldots l-1}$);

- there are sorts s_i, for $1 \leq i \leq k$ such that, for $\Gamma_{I'} = [I_1 : \forall \Gamma_{P'}, (A_1)_{/s_1}; \ldots; I_k : \forall \Gamma_{P'}, (A_k)_{/s_k}]$ we have $(E[\Gamma; \Gamma_{I'}; \Gamma_{P'}] \vdash C_i : s_{q_i})_{i=1\ldots n}$;
- the sorts are such that all eliminations, to Prop, Set and Type(j), are allowed (see section 4.5.4).

Notice that if $I_j\ q_1\ \ldots\ q_r$ is typable using the rules **Ind-Const** and **App**, then it is typable using the rule **Ind-Family**. Conversely, the extended theory is not stronger than the theory without **Ind-Family**. We get an equiconsistency result by mapping each $\mathsf{Ind}(\Gamma)[p](\Gamma_I := \Gamma_C)$ occurring into a given derivation into as many different inductive types and constructors as the number of different (partial) replacements of sorts, needed for this derivation, in the parameters that are arities (this is possible because $\mathsf{Ind}(\Gamma)[p](\Gamma_I := \Gamma_C)$ well-formed implies that $\mathsf{Ind}(\Gamma)[p](\Gamma_{I'} := \Gamma_{C'})$ is well-formed and has the same allowed eliminations, where $\Gamma_{I'}$ is defined as above and $\Gamma_{C'} = [c_1 : \forall \Gamma_{P'}, C_1; \ldots; c_n : \forall \Gamma_{P'}, C_n]$). That is, the changes in the types of each partial instance $q_1\ \ldots\ q_r$ can be characterized by the ordered sets of arity sorts among the types of parameters, and to each signature is associated a new inductive definition with fresh names. Conversion is preserved as any (partial) instance $I_j\ q_1\ \ldots\ q_r$ or $C_i\ q_1\ \ldots\ q_r$ is mapped to the names chosen in the specific instance of $\mathsf{Ind}(\Gamma)[p](\Gamma_I := \Gamma_C)$.

In practice, the rule **Ind-Family** is used by Coq only when all the inductive types of the inductive definition are declared with an arity whose sort is in the Type hierarchy. Then, the polymorphism is over the parameters whose type is an arity of sort in the Type hierarchy. The sort s_j are chosen canonically so that each s_j is minimal with respect to the hierarchy Prop $\subset$ Set$_p$ $\subset$ Type where Set$_p$ is predicative Set. More precisely, an empty or small singleton inductive definition (i.e. an inductive definition of which all inductive types are singleton – see paragraph 4.5.4) is set in Prop, a small non-singleton inductive family is set in Set (even in case Set is impredicative – see Section 4.7), and otherwise in the Type hierarchy.

Note that the side-condition about allowed elimination sorts in the rule **Ind-Family** is just to avoid to recompute the allowed elimination sorts at each instance of a pattern-matching (see section 4.5.4).

As an example, let us consider the following definition:

```
Coq < Inductive option (A:Type) : Type :=
Coq < | None : option A
Coq < | Some : A -> option A.
```

As the definition is set in the Type hierarchy, it is used polymorphically over its parameters whose types are arities of a sort in the Type hierarchy. Here, the parameter A has this property, hence, if `option` is applied to a type in Set, the result is in Set. Note that if `option` is applied to a type in Prop, then, the result is not set in `Prop` but in `Set` still. This is because `option` is not a singleton type (see section 4.5.4) and it would loose the elimination to Set and Type if set in Prop.

```
Coq < Check (fun A:Set => option A).
fun A : Set => option A
     : Set -> Set

Coq < Check (fun A:Prop => option A).
fun A : Prop => option A
     : Prop -> Set
```

Here is another example.

```
Coq < Inductive prod (A B:Type) : Type := pair : A -> B -> prod A B.
```

As `prod` is a singleton type, it will be in Prop if applied twice to propositions, in Set if applied twice to at least one type in Set and none in Type, and in Type otherwise. In all cases, the three kind of eliminations schemes are allowed.

```
Coq < Check (fun A:Set => prod A).
fun A : Set => prod A
     : Set -> Type -> Type

Coq < Check (fun A:Prop => prod A A).
fun A : Prop => prod A A
     : Prop -> Prop

Coq < Check (fun (A:Prop) (B:Set) => prod A B).
fun (A : Prop) (B : Set) => prod A B
     : Prop -> Set -> Set

Coq < Check (fun (A:Type) (B:Prop) => prod A B).
fun (A : Type) (B : Prop) => prod A B
     : Type -> Prop -> Type
```

4.5.4 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 normalizing 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 : Set exists in the context, we want to build a function length of type List_A $\rightarrow$ nat which computes the length of the list, so such that $(\mathsf{length}\ (\mathsf{nil}\ A)) = \mathsf{O}$ and $(\mathsf{length}\ (\mathsf{cons}\ A\ a\ l)) = (\mathsf{S}\ (\mathsf{length}\ l))$. We want these equalities to be recognized implicitly 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 $\forall l$: List_A, $(\mathsf{Length\_A}\ l\ (\mathsf{length}\ l))$ it is enough to prove:
$(\mathsf{Length\_A}\ (\mathsf{nil}\ A)\ (\mathsf{length}\ (\mathsf{nil}\ A)))$ and

$\forall a : A, \forall l$: List_A, $(\mathsf{Length\_A}\ l\ (\mathsf{length}\ l)) \rightarrow (\mathsf{Length\_A}\ (\mathsf{cons}\ A\ a\ l)\ (\mathsf{length}\ (\mathsf{cons}\ A\ a\ l)))$.

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

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 [27]. One is the definition by pattern-matching. The second one is a definition by guarded fixpoints.

The `match...with ...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 $m = (c_i\ u_1 \ldots u_{p_i})$ for each constructor of I.

The Coq term for this proof will be written :

$$\mathsf{match}\ m\ \mathsf{with}\ (c_1\ x_{11}\ ...\ x_{1p_1}) \Rightarrow f_1 \mid \ldots \mid (c_n\ x_{n1}...x_{np_n}) \Rightarrow f_n\ \mathsf{end}$$

In this expression, if m is a term built from a constructor $(c_i\ u_1 \ldots u_{p_i})$ then the expression will behave as it is specified with i-th branch and will reduce to f_i where the $x_{i1} \ldots x_{ip_i}$ are replaced by the $u_1 \ldots u_p$ according to the ι-reduction.

Actually, for type-checking a match...with...end expression we also need to know the predicate P to be proved by case analysis. In the general case where I is an inductively defined n-ary relation, P is a $n+1$-ary relation: the n first arguments correspond to the arguments of I (parameters excluded), and the last one corresponds to object m. Coq can sometimes infer this predicate but sometimes not. The concrete syntax for describing this predicate uses the as...in...return construction. For instance, let us assume that I is an unary predicate with one parameter. The predicate is made explicit using the syntax :

$$\mathsf{match}\ m\ \mathsf{as}\ x\ \mathsf{in}\ I\ \_\ a\ \mathsf{return}\ (P\ x)\ \mathsf{with}\ (c_1\ x_{11}\ ...\ x_{1p_1}) \Rightarrow f_1 \mid \ldots \mid (c_n\ x_{n1}...x_{np_n}) \Rightarrow f_n \mathsf{end}$$

The as part can be omitted if either the result type does not depend on m (non-dependent elimination) or m is a variable (in this case, the result type can depend on m). The in part can be omitted if the result type does not depend on the arguments of I. Note that the arguments of I corresponding to parameters *must* be _, because the result type is not generalized to all possible values of the parameters. The expression after in must be seen as an *inductive type pattern*. As a final remark, expansion of implicit arguments and notations apply to this pattern.

For the purpose of presenting the inference rules, we use a more compact notation :

$$\mathsf{case}(m, (\lambda a x, P), \lambda x_{11}\ ...\ x_{1p_1}, f_1 \mid \ldots \mid \lambda x_{n1}...x_{np_n}, f_n)$$

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

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

The case of inductive definitions in sorts Set or Type is simple. There is no restriction on the sort of the predicate to be eliminated.

Notations. The $[I : A|B]$ is defined as the smallest relation satisfying the following rules: We write $[I|B]$ for $[I : A|B]$ where A is the type of I.

Prod

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

Set& Type

$$\frac{s_1 \in \{\mathsf{Set}, \mathsf{Type}(j)\}, s_2 \in \mathcal{S}}{[I : s_1|I \to s_2]}$$

The case of Inductive definitions of sort Prop is a bit more complicated, because of our interpretation of this sort. The only harmless allowed elimination, is the one when predicate P is also of sort Prop.

Prop

$$[I : \mathsf{Prop}|I \to \mathsf{Prop}]$$

Prop is the type of logical propositions, the proofs of properties P in Prop could not be used for computation and are consequently ignored by the extraction mechanism. Assume A and B are two propositions, and the logical disjunction $A \vee B$ is defined inductively by :

```
Coq < Inductive or (A B:Prop) : Prop :=
Coq <   lintro : A -> or A B | rintro : B -> or A B.
```

The following definition which computes a boolean value by case over the proof of `or A B` is not accepted :

```
Coq < Definition choice (A B: Prop) (x:or A B) :=
Coq <    match x with lintro a => true | rintro b => false end.
Coq < Coq < Error:
Incorrect elimination of "x" in the inductive type "or":
the return type has sort "Set" while it should be "Prop".
Elimination of an inductive object of sort Prop
is not allowed on a predicate in sort Set
because proofs can be eliminated only to build proofs.
```

From the computational point of view, the structure of the proof of `(or A B)` in this term is needed for computing the boolean value.

In general, if I has type Prop then P cannot have type $I \to$ Set, because it will mean to build an informative proof of type $(P\ m)$ doing a case analysis over a non-computational object that will disappear in the extracted program. 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.

In the same spirit, elimination on P of type $I \to$ Type cannot be allowed because it trivially implies the elimination on P of type $I \to$ Set by cumulativity. It also implies that there is two proofs of the same property which are provably different, contradicting the proof-irrelevance property which is sometimes a useful axiom :

```
Coq < Axiom proof_irrelevance : forall (P : Prop) (x y : P), x=y.
proof_irrelevance is assumed
```

The elimination of an inductive definition of type Prop on a predicate P of type $I \to$ Type leads to a paradox when applied to impredicative inductive definition like the second-order existential quantifier `exProp` defined above, because it give access to the two projections on this type.

Empty and singleton elimination There are special inductive definitions in Prop for which more eliminations are allowed.

Prop-extended

$$\frac{I \text{ is an empty or singleton definition} \quad s \in \mathcal{S}}{[I : \mathsf{Prop} | I \to s]}$$

A *singleton definition* has only one constructor and all the arguments of this constructor have type Prop. In that case, there is a canonical way to interpret the informative extraction on an object in that type, such that the elimination on any sort s is legal. Typical examples are the conjunction of non-informative propositions and the equality. If there is an hypothesis $h : a = b$ in the context, it can be used for rewriting not only in logical propositions but also in any type.

```
Coq < Print eq_rec.
eq_rec =
fun (A : Type) (x : A) (P : A -> Set) => eq_rect x P
     : forall (A : Type) (x : A) (P : A -> Set),
       P x -> forall y : A, x = y -> P y
Argument A is implicit
Argument scopes are [type_scope _ _ _ _ _]

Coq < Extraction eq_rec.
(** val eq_rec : 'a1 -> 'a2 -> 'a1 -> 'a2 **)
let eq_rec x f y =
  f
```

An empty definition has no constructors, in that case also, elimination on any sort is allowed.

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.

$$\begin{aligned} \{c : (I_i\ p_1 \dots p_r\ t_1 \dots t_p)\}^P &\equiv (P\ t_1 \dots\ t_p\ c) \\ \{c : \forall\ x : T, C\}^P &\equiv \forall\ x : T, \{(c\ x) : C\}^P \end{aligned}$$

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

Examples. For List_A the type of P will be List_A $\to s$ for $s \in \mathcal{S}$.
$\{(\mathsf{cons}\ A)\}^P \equiv \forall a : A, \forall l : \mathsf{List\_A}, (P\ (\mathsf{cons}\ A\ a\ l))$.

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

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

match

$$\frac{E[\Gamma] \vdash c : (I\ q_1 \dots q_r\ t_1 \dots t_s)\ \ E[\Gamma] \vdash P : B\ \ [(I\ q_1 \dots q_r)|B]\ \ (E[\Gamma] \vdash f_i : \{(c_{p_i}\ q_1 \dots q_r)\}^P)_{i=1\dots l}}{E[\Gamma] \vdash \mathsf{case}(c, P, f_1| \dots |f_l) : (P\ t_1 \dots t_s\ c)}$$

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

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

$$\frac{l : \mathsf{List\_A} \;\; P : \mathsf{List\_A} \to s \;\; f_1 : (P\ (\mathsf{nil}\ A)) \;\; f_2 : \forall a : A, \forall l : \mathsf{List\_A}, (P\ (\mathsf{cons}\ A\ a\ l))}{\mathsf{case}(l, P, f_1 \mid f_2) : (P\ l)}$$

$$\frac{\begin{array}{c} H : (\mathsf{Length\_A}\ L\ N) \\ P : \forall l : \mathsf{List\_A}, \forall n : \mathsf{nat}, (\mathsf{Length\_A}\ l\ n) \to \mathsf{Prop} \\ f_1 : (P\ (\mathsf{nil}\ A)\ \mathsf{O}\ \mathsf{Lnil}) \\ f_2 : \forall a : A, \forall l : \mathsf{List\_A}, \forall n : \mathsf{nat}, \forall h : (\mathsf{Length\_A}\ l\ n), (P\ (\mathsf{cons}\ A\ a\ n)\ (\mathsf{S}\ n)\ (\mathsf{Lcons}\ A\ a\ l\ n\ h)) \end{array}}{\mathsf{case}(H, P, f_1 \mid f_2) : (P\ L\ N\ H)}$$

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:

$$\mathsf{case}((c_{p_i}\ q_1 \ldots q_r\ a_1 \ldots a_m), P, f_1 | \ldots | f_l)$$

with c_{p_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:

$$\mathsf{case}((c_{p_i}\ q_1 \ldots q_r\ a_1 \ldots a_m), P, f_1 | \ldots | f_n) \triangleright_\iota (f_i\ a_1 \ldots a_m)$$

4.5.5 Fixpoint definitions

The second operator for elimination is fixpoint definition. This fixpoint may involve several mutually recursive definitions. The basic concrete syntax for a recursive set of mutually recursive declarations is (with Γ_i contexts) :

$$\mathsf{fix}\ f_1(\Gamma_1) : A_1 := t_1\ \mathsf{with} \ldots \mathsf{with}\ f_n(\Gamma_n) : A_n := t_n$$

The terms are obtained by projections from this set of declarations and are written

$$\mathsf{fix}\ f_1(\Gamma_1) : A_1 := t_1\ \mathsf{with} \ldots \mathsf{with}\ f_n(\Gamma_n) : A_n := t_n\ \mathsf{for}\ f_i$$

In the inference rules, we represent such a term by

$$\mathtt{Fix}\ f_i\{f_1 : A'_1 := t'_1 \ldots f_n : A'_n := t'_n\}$$

with t'_i (resp. A'_i) representing the term t_i abstracted (resp. generalized) with respect to the bindings in the context Γ_i, namely $t'_i = \lambda\Gamma_i, t_i$ and $A'_i = \forall\Gamma_i, A_i$.

Typing rule

The typing rule is the expected one for a fixpoint.

Fix

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

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 a 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

$$\forall P : \mathsf{nat} \to \mathsf{Prop}, (P\ \mathsf{O}) \to (\forall n : \mathsf{nat}, (P\ n) \to (P\ (\mathsf{S}\ n))) \to \forall n : \mathsf{nat}, (P\ n)$$

can be represented by the term:

$$\lambda P : \mathsf{nat} \to \mathsf{Prop}, \lambda f : (P\ \mathsf{O}), \lambda g : (\forall n : \mathsf{nat}, (P\ n) \to (P\ (\mathsf{S}\ n))),$$
$$\texttt{Fix}\ h\{h : \forall n : \mathsf{nat}, (P\ n) := \lambda n : \mathsf{nat}, \mathsf{case}(n, P, f \mid \lambda p : \mathsf{nat}, (g\ p\ (h\ p)))\}$$

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 [65].

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

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

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

Now in the definition t_i, if f_j occurs then it should be applied to at least k_j arguments and 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. One needs first to define the notion of *recursive arguments of a constructor*. For an inductive definition $\mathsf{Ind}(\Gamma)[r](\Gamma_I := \Gamma_C)$, the type of a constructor c has the form $\forall p_1 : P_1, \ldots \forall p_r : P_r, \forall x_1 : T_1, \ldots \forall x_r : T_r, (I_j\ p_1 \ldots p_r\ t_1 \ldots t_s)$ the recursive arguments will correspond to T_i in which one of the I_l occurs.

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

- $(t\ u), \lambda x : u, t$ when t is structurally smaller than y .
- $\mathsf{case}(c, P, f_1 \ldots f_n)$ 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 $C_q \equiv \forall p_1 : P_1, \ldots, \forall p_r : P_r, \forall y_1 : B_1, \ldots \forall y_k : B_k, (I\ a_1 \ldots a_k)$ and can consequently be written $\lambda y_1 : B'_1, \ldots \lambda y_k : B'_k, g_i$. ($B'_i$ 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 1.3.4 and show the internal representation.

```
Coq < Fixpoint plus (n m:nat) {struct n} : nat :=
Coq <   match n with
Coq <   | O => m
Coq <   | S p => S (plus p m)
Coq <   end.
plus is recursively defined (decreasing on 1st argument)

Coq < Print plus.
plus =
fix plus (n m : nat) : nat :=
  match n with
  | O => m
  | S p => S (plus p m)
  end
     : nat -> nat -> nat

Coq < Fixpoint lgth (A:Set) (l:list A) {struct l} : nat :=
Coq <   match l with
Coq <   | nil => O
Coq <   | cons a l' => S (lgth A l')
Coq <   end.
lgth is recursively defined (decreasing on 2nd argument)

Coq < Print lgth.
lgth =
fix lgth (A : Set) (l : list A) {struct l} : nat :=
  match l with
  | nil => O
  | cons _ l' => S (lgth A l')
  end
     : forall A : Set, list A -> nat
Argument scopes are [type_scope _]

Coq < Fixpoint sizet (t:tree) : nat := let (f) := t in S (sizef f)
Coq <  with sizef (f:forest) : nat :=
Coq <   match f with
Coq <   | emptyf => O
Coq <   | consf t f => plus (sizet t) (sizef f)
Coq <   end.
sizet, sizef are recursively defined (decreasing respectively on 1st,
1st arguments)

Coq < Print sizet.
sizet =
fix sizet (t : tree) : nat :=
  let (f) := t in S (sizef f)
with sizef (f : forest) : nat :=
  match f with
  | emptyf => O
  | consf t f0 => plus (sizet t) (sizef f0)
  end
for sizet
     : tree -> nat
```

Reduction rule

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

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

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 forall n m:nat, plus (S n) m = S (plus n m).
1 subgoal

  ============================
   forall n m : nat, plus (S n) m = S (plus n m)

Coq < reflexivity.
Proof completed.

Coq < Abort.
Current goal aborted

Coq < Goal forall f:forest, sizet (node f) = S (sizef f).
1 subgoal

  ============================
   forall f : forest, sizet (node f) = S (sizef f)

Coq < reflexivity.
Proof completed.

Coq < Abort.
Current goal aborted
```

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

```
Coq < Definition sont (t:tree) : forest
Coq <    := let (f) := t in f.
sont is defined
```

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

```
Coq < Goal forall t:tree, sizet t = S (sizef (sont t)).
Coq < Coq < 1 subgoal

  ============================
   forall t : tree, sizet t = S (sizef (sont t))

Coq < reflexivity. (** this one fails **)
Toplevel input, characters 0-11:
> reflexivity.
> ^^^^^^^^^^^
Error: Impossible to unify "S (sizef (sont t))" with "sizet t".

Coq < destruct t.
1 subgoal

  f : forest
  ============================
   sizet (node f) = S (sizef (sont (node f)))

Coq < reflexivity.
Proof completed.
```

Mutual induction

The principles of mutual induction can be automatically generated using the `Scheme` command described in Section 8.14.

4.6 Coinductive types

The implementation contains also coinductive definitions, which are types inhabited by infinite objects. More information on coinductive definitions can be found in [66, 67, 68].

4.7 CIC: the Calculus of Inductive Construction with impredicative Set

COQ can be used as a type-checker for CIC, the original Calculus of Inductive Constructions with an impredicative sort Set by using the compiler option `-impredicative-set`.

For example, using the ordinary `coqtop` command, the following is rejected.

```
Coq < Definition id: Set := forall X:Set,X->X.
Coq < Coq < Coq < Coq < Toplevel input, characters 185-202:
> Definition id: Set := forall X:Set,X->X.
>                       ^^^^^^^^^^^^^^^^^
Error: The term "forall X : Set, X -> X" has type "Type"
 while it is expected to have type "Set".
```

while it will type-check, if one use instead the `coqtop -impredicative-set` command.

The major change in the theory concerns the rule for product formation in the sort Set, which is extended to a domain in any sort :

Prod

$$\frac{E[\Gamma] \vdash T : s \quad s \in \mathcal{S} \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{Set}}{E[\Gamma] \vdash \forall\, x : T, U : \mathsf{Set}}$$

This extension has consequences on the inductive definitions which are allowed. In the impredicative system, one can build so-called *large inductive definitions* like the example of second-order existential quantifier (`exSet`).

There should be restrictions on the eliminations which can be performed on such definitions. The eliminations rules in the impredicative system for sort Set become :

Set

$$\frac{s \in \{\mathsf{Prop}, \mathsf{Set}\}}{[I : \mathsf{Set} | I \to s]} \qquad \frac{I \text{ is a small inductive definition} \quad s \in \{\mathsf{Type}(i)\}}{[I : \mathsf{Set} | I \to s]}$$

Chapter 5

The Module System

The module system extends the Calculus of Inductive Constructions providing a convenient way to structure large developments as well as a mean of massive abstraction.

5.1 Modules and module types

Access path. It is denoted by p, it can be either a module variable X or, if p' is an access path and id an identifier, then $p'.id$ is an access path.

Structure element. It is denoted by e and is either a definition of a constant, an assumption, a definition of an inductive, a definition of a module, an alias of module or a module type abbreviation.

Structure expression. It is denoted by S and can be:

- an access path p
- a plain structure $\mathsf{Struct}\ e;\ldots;e\ \mathsf{End}$
- a functor $\mathsf{Functor}(X : S)\ S'$, where X is a module variable, S and S' are structure expression
- an application $S\,p$, where S is a structure expression and p an access path
- a refined structure S with $p := p'$ or S with $p := t : T$ where S is a structure expression, p and p' are access paths, t is a term and T is the type of t.

The symbol W will be used to denote a plain structure or a functor.

Module definition, is written $\mathsf{Mod}(X : S\,[:= S'])$ and consists of a module variable X, a module type S which can be any structure expression and optionally a module implementation S' which can be any structure expression except a refined structure.

Module alias, is written $\mathsf{ModA}(X == p)$ and consists of a module variable X and a module path p.

Module type abbreviation, is written $\mathsf{ModType}(Y := S)$, where Y is an identifier and S is any structure expression .

5.2 Typing Modules

In order to introduce the typing system we first slightly extend the syntactic class of terms and environments given in section 4.1. The environments, apart from definitions of constants and inductive types now also hold any other structure elements. Terms, apart from variables, constants and complex terms, include also access paths.

We also need additional typing judgments:

- $E[] \vdash \mathcal{WF}(S)$, denoting that a structure S is well-formed,
- $E[] \vdash p : S$, denoting that the module pointed by p has type S in environment E.
- $E[] \vdash S \longrightarrow W$, denoting that a structure S is evaluated to a structure W in weak head normal form.
- $E[] \vdash S_1 <: S_2$, denoting that a structure S_1 is a subtype of a structure S_2.
- $E[] \vdash e_1 <: e_2$, denoting that a structure element e_1 is more precise that a structure element e_2.

The rules for forming structures are the following:

WF-STR

$$\frac{\mathcal{WF}(E; E')[]}{E[] \vdash \mathcal{WF}(\mathsf{Struct}\ E'\ \mathsf{End})}$$

WF-FUN

$$\frac{E; \mathsf{Mod}(X : S)[] \vdash S' \longrightarrow W \qquad E; \mathsf{Mod}(X : S)[] \vdash \mathcal{WF}(W)}{E[] \vdash \mathcal{WF}(\mathsf{Functor}(X : S)\ S')}$$

Evaluation of structures to weak head normal form:

WEVAL-APP

$$\frac{\begin{array}{c} E[] \vdash S \longrightarrow \mathsf{Functor}(X : S_1)\ S_2 \quad E[] \vdash S_1 \longrightarrow W_1 \\ E[] \vdash p : W_3 \qquad E[] \vdash W_3 <: W_1 \end{array}}{E[] \vdash S\,p \longrightarrow S_2\{p/X, t_1/p_1.c_1, \ldots, t_n/p_n.c_n\}}$$

In the last rule, $\{t_1/p_1.c_1, \ldots, t_n/p_n.c_n\}$ is the resulting substitution from the inlining mechanism. We substitute in S the inlined fields $p_i.c_i$ form $\mathsf{Mod}(X : S_1)$ by the corresponding delta-reduced term t_i in p.

WEVAL-WITH-MOD

$$\frac{\begin{array}{c} E[] \vdash S \longrightarrow \mathsf{Struct}\ e_1; \ldots; e_{i-1}; \mathsf{Mod}(X : S_1); e_{i+2}; \ldots; e_n\ \mathsf{End} \quad E; e_1; \ldots; e_{i-1}[] \vdash S_1 \longrightarrow W_1 \\ E[] \vdash p : W_2 \qquad E; e_1; \ldots; e_{i-1}[] \vdash W_2 <: W_1 \end{array}}{\begin{array}{c} E[] \vdash S\ \mathsf{with}\ x := p \longrightarrow \\ \mathsf{Struct}\ e_1; \ldots; e_{i-1}; \mathsf{ModA}(X_1 == p); e_{i+2}\{p/X\}; \ldots; e_n\{p/X\}\ \mathsf{End} \end{array}}$$

WEVAL-WITH-MOD-REC

$$\frac{\begin{array}{c} E[] \vdash S \longrightarrow \mathsf{Struct}\ e_1; \ldots; e_{i-1}; \mathsf{Mod}(X_1 : S_1); e_{i+2}; \ldots; e_n\ \mathsf{End} \\ E; e_1; \ldots; e_{i-1}[] \vdash S_1\ \mathsf{with}\ p := p_1 \longrightarrow W_1 \end{array}}{\begin{array}{c} E[] \vdash S\ \mathsf{with}\ X_1.p := p_1 \longrightarrow \\ \mathsf{Struct}\ e_1; \ldots; e_{i-1}; \mathsf{Mod}(X_1 : W_1); e_{i+2}\{p_1/X_1.p\}; \ldots; e_n\{p_1/X_1.p\}\ \mathsf{End} \end{array}}$$

WEVAL-WITH-DEF

$$\frac{E[] \vdash S \longrightarrow \mathsf{Struct}\ e_1; \ldots; e_{i-1}; \mathsf{Assum}()(c : T_1); e_{i+2}; \ldots; e_n\ \mathsf{End} \quad E; e_1; \ldots; e_{i-1}[] \vdash \mathsf{Def}()(c := t : T) <: \mathsf{Assum}()(c : T_1)}{E[] \vdash S\ \mathsf{with}\ c := t : T \longrightarrow \mathsf{Struct}\ e_1; \ldots; e_{i-1}; \mathsf{Def}()(c := t : T); e_{i+2}; \ldots; e_n\ \mathsf{End}}$$

WEVAL-WITH-DEF-REC

$$\frac{E[] \vdash S \longrightarrow \mathsf{Struct}\ e_1; \ldots; e_{i-1}; \mathsf{Mod}(X_1 : S_1); e_{i+2}; \ldots; e_n\ \mathsf{End} \quad E; e_1; \ldots; e_{i_1}[] \vdash S_1\ \mathsf{with}\ p := p_1 \longrightarrow W_1}{E[] \vdash S\ \mathsf{with}\ X_1.p := t : T \longrightarrow \mathsf{Struct}\ e_1; \ldots; e_{i-1}; \mathsf{Mod}(X : W_1); e_{i+2}; \ldots; e_n\ \mathsf{End}}$$

WEVAL-PATH-MOD

$$\frac{E[] \vdash p \longrightarrow \mathsf{Struct}\ e_1; \ldots; e_{i-1}; \mathsf{Mod}(X : S\,[:= S_1]); e_{i+2}; \ldots; e_n\ \mathsf{End} \quad E; e_1; \ldots; e_{i-1}[] \vdash S \longrightarrow W}{E[] \vdash p.X \longrightarrow W}$$

$$\frac{\mathcal{WF}(E)[] \qquad \mathsf{Mod}(X : S\,[:= S_1]) \in E \quad E[] \vdash S \longrightarrow W}{E[] \vdash X \longrightarrow W}$$

WEVAL-PATH-ALIAS

$$\frac{E[] \vdash p \longrightarrow \mathsf{Struct}\ e_1; \ldots; e_{i-1}; \mathsf{ModA}(X == p_1); e_{i+2}; \ldots; e_n\ \mathsf{End} \quad E; e_1; \ldots; e_{i-1}[] \vdash p_1 \longrightarrow W}{E[] \vdash p.X \longrightarrow W}$$

$$\frac{\mathcal{WF}(E)[] \qquad \mathsf{ModA}(X == p_1) \in E \quad E[] \vdash p_1 \longrightarrow W}{E[] \vdash X \longrightarrow W}$$

WEVAL-PATH-TYPE

$$\frac{E[] \vdash p \longrightarrow \mathsf{Struct}\ e_1; \ldots; e_{i-1}; \mathsf{ModType}(Y := S); e_{i+2}; \ldots; e_n\ \mathsf{End} \quad E; e_1; \ldots; e_{i-1}[] \vdash S \longrightarrow W}{E[] \vdash p.Y \longrightarrow W}$$

WEVAL-PATH-TYPE

$$\frac{\mathcal{WF}(E)[] \qquad \mathsf{ModType}(Y := S) \in E \quad E[] \vdash S \longrightarrow W}{E[] \vdash Y \longrightarrow W}$$

Rules for typing module:

MT-EVAL-STR

$$\frac{E[] \vdash p \longrightarrow W}{E[] \vdash p : W/p}$$

The last rule, called strengthening is used to make all module fields manifestly equal to themselves. The notation W/p has the following meaning:

- if $W \longrightarrow \mathsf{Struct}\ e_1;\ldots;e_n\ \mathsf{End}$ then $W/p = \mathsf{Struct}\ e_1/p;\ldots;e_n/p\ \mathsf{End}$ where e/p is defined as follows:
  - $\mathsf{Def}()(c := t : T)/p$[1] $= \mathsf{Def}()(c := t : T)$
  - $\mathsf{Assum}()(c : U)/p = \mathsf{Def}()(c := p.c : U)$
  - $\mathsf{Mod}(X : S)/p = \mathsf{ModA}(X == p.X)$
  - $\mathsf{ModA}(X == p')/p = \mathsf{ModA}(X == p')$
  - $\mathsf{Ind}()[\Gamma_P](\Gamma_C := \Gamma_I)/p = \mathsf{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I)$
  - $\mathsf{Ind}_{p'}()[\Gamma_P](\Gamma_C := \Gamma_I)/p = \mathsf{Ind}_{p'}()[\Gamma_P](\Gamma_C := \Gamma_I)$
- if $W \longrightarrow \mathsf{Functor}(X : S')\ S''$ then $W/p = W$

The notation $\mathsf{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I)$ denotes an inductive definition that is definitionally equal to the inductive definition in the module denoted by the path p. All rules which have $\mathsf{Ind}()[\Gamma_P](\Gamma_C := \Gamma_I)$ as premises are also valid for $\mathsf{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I)$. We give the formation rule for $\mathsf{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I)$ below as well as the equality rules on inductive types and constructors.

The module subtyping rules:

MSUB-STR

$$\frac{E; e_1;\ldots;e_n[] \vdash e_{\sigma(i)} <: e'_i \text{ for } i = 1..m \qquad \sigma : \{1\ldots m\} \to \{1\ldots n\} \text{ injective}}{E[] \vdash \mathsf{Struct}\ e_1;\ldots;e_n\ \mathsf{End} <: \mathsf{Struct}\ e'_1;\ldots;e'_m\ \mathsf{End}}$$

MSUB-FUN

$$\frac{\begin{array}{cc} E[] \vdash S_1 \longrightarrow W_1 & E[] \vdash S'_1 \longrightarrow W'_1 \\ E; \mathsf{Mod}(X : S_1)[] \vdash S_2 \longrightarrow W_2 & E; \mathsf{Mod}(X : S'_1)[] \vdash S'_2 \longrightarrow W'_2 \\ E[] \vdash W'_1 <: W_1 & E; \mathsf{Mod}(X : S'_1)[] \vdash W_2 <: W'_2 \end{array}}{E[] \vdash \mathsf{Functor}(X : S_1)\ S_2 <: \mathsf{Functor}(X : S'_1)\ S'_2}$$

Structure element subtyping rules:

ASSUM-ASSUM

$$\frac{E[] \vdash T_1 \leq_{\beta\delta\iota\zeta} T_2}{E[] \vdash \mathsf{Assum}()(c : T_1) <: \mathsf{Assum}()(c : T_2)}$$

DEF-ASSUM

$$\frac{E[] \vdash T_1 \leq_{\beta\delta\iota\zeta} T_2}{E[] \vdash \mathsf{Def}()(c := t : T_1) <: \mathsf{Assum}()(c : T_2)}$$

ASSUM-DEF

$$\frac{E[] \vdash T_1 \leq_{\beta\delta\iota\zeta} T_2 \qquad E[] \vdash c =_{\beta\delta\iota\zeta} t_2}{E[] \vdash \mathsf{Assum}()(c : T_1) <: \mathsf{Def}()(c := t_2 : T_2)}$$

[1] Opaque definitions are processed as assumptions.

DEF-DEF

$$\frac{E[] \vdash T_1 \leq_{\beta\delta\iota\zeta} T_2 \qquad E[] \vdash t_1 =_{\beta\delta\iota\zeta} t_2}{E[] \vdash \mathsf{Def}()(c := t_1 : T_1) <: \mathsf{Def}()(c := t_2 : T_2)}$$

IND-IND

$$\frac{E[] \vdash \Gamma_P =_{\beta\delta\iota\zeta} \Gamma'_P \qquad E[\Gamma_P] \vdash \Gamma_C =_{\beta\delta\iota\zeta} \Gamma'_C \qquad E[\Gamma_P;\Gamma_C] \vdash \Gamma_I =_{\beta\delta\iota\zeta} \Gamma'_I}{E[] \vdash \mathsf{Ind}()[\Gamma_P](\Gamma_C := \Gamma_I\,) <: \mathsf{Ind}()[\Gamma'_P](\Gamma'_C := \Gamma'_I\,)}$$

INDP-IND

$$\frac{E[] \vdash \Gamma_P =_{\beta\delta\iota\zeta} \Gamma'_P \qquad E[\Gamma_P] \vdash \Gamma_C =_{\beta\delta\iota\zeta} \Gamma'_C \qquad E[\Gamma_P;\Gamma_C] \vdash \Gamma_I =_{\beta\delta\iota\zeta} \Gamma'_I}{E[] \vdash \mathsf{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I\,) <: \mathsf{Ind}()[\Gamma'_P](\Gamma'_C := \Gamma'_I\,)}$$

INDP-INDP

$$\frac{E[] \vdash \Gamma_P =_{\beta\delta\iota\zeta} \Gamma'_P \qquad E[\Gamma_P] \vdash \Gamma_C =_{\beta\delta\iota\zeta} \Gamma'_C \qquad E[\Gamma_P;\Gamma_C] \vdash \Gamma_I =_{\beta\delta\iota\zeta} \Gamma'_I \qquad E[] \vdash p =_{\beta\delta\iota\zeta} p'}{E[] \vdash \mathsf{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I\,) <: \mathsf{Ind}_{p'}()[\Gamma'_P](\Gamma'_C := \Gamma'_I\,)}$$

MOD-MOD

$$\frac{E[] \vdash S_1 <: S_2}{E[] \vdash \mathsf{Mod}(X : S_1) <: \mathsf{Mod}(X : S_2)}$$

ALIAS-MOD

$$\frac{E[] \vdash p : S_1 \qquad E[] \vdash S_1 <: S_2}{E[] \vdash \mathsf{ModA}(X == p) <: \mathsf{Mod}(X : S_2)}$$

MOD-ALIAS

$$\frac{E[] \vdash p : S_2 \qquad E[] \vdash S_1 <: S_2 \qquad E[] \vdash X =_{\beta\delta\iota\zeta} p}{E[] \vdash \mathsf{Mod}(X : S_1) <: \mathsf{ModA}(X == p)}$$

ALIAS-ALIAS

$$\frac{E[] \vdash p_1 =_{\beta\delta\iota\zeta} p_2}{E[] \vdash \mathsf{ModA}(X == p_1) <: \mathsf{ModA}(X == p_2)}$$

MODTYPE-MODTYPE

$$\frac{E[] \vdash S_1 <: S_2 \qquad E[] \vdash S_2 <: S_1}{E[] \vdash \mathsf{ModType}(Y := S_1) <: \mathsf{ModType}(Y := S_2)}$$

New environment formation rules

WF-MOD

$$\frac{\mathcal{WF}(E)[] \qquad E[] \vdash \mathcal{WF}(S)}{\mathcal{WF}(E;\mathsf{Mod}(X : S))[]}$$

WF-MOD

$$\frac{\begin{array}{c}E[] \vdash S_2 <: S_1 \\ \mathcal{WF}(E)[] \quad E[] \vdash \mathcal{WF}(S_1) \quad E[] \vdash \mathcal{WF}(S_2)\end{array}}{\mathcal{WF}(E;\mathsf{Mod}(X : S_1\,[:= S_2]))[]}$$

WF-ALIAS

$$\frac{\mathcal{WF}(E)[] \qquad E[] \vdash p : S}{\mathcal{WF}(E,\mathsf{ModA}(X == p))[]}$$

WF-MODTYPE

$$\frac{\mathcal{WF}(E)[] \qquad E[] \vdash \mathcal{WF}(S)}{\mathcal{WF}(E, \mathsf{ModType}(Y := S))[]}$$

WF-IND

$$\frac{\begin{array}{c}\mathcal{WF}(E; \mathsf{Ind}()[\Gamma_P](\Gamma_C := \Gamma_I))[] \\ E[] \vdash p : \mathsf{Struct}\ e_1; \ldots; e_i; \mathsf{Ind}()[\Gamma'_P](\Gamma'_C := \Gamma'_I); \ldots\ \mathsf{End} : \\ E[] \vdash \mathsf{Ind}()[\Gamma'_P](\Gamma'_C := \Gamma'_I) <: \mathsf{Ind}()[\Gamma_P](\Gamma_C := \Gamma_I)\end{array}}{\mathcal{WF}(E; \mathsf{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I))[]}$$

Component access rules

ACC-TYPE

$$\frac{E[\Gamma] \vdash p : \mathsf{Struct}\ e_1; \ldots; e_i; \mathsf{Assum}()(c : T); \ldots\ \mathsf{End}}{E[\Gamma] \vdash p.c : T}$$

$$\frac{E[\Gamma] \vdash p : \mathsf{Struct}\ e_1; \ldots; e_i; \mathsf{Def}()(c := t : T); \ldots\ \mathsf{End}}{E[\Gamma] \vdash p.c : T}$$

ACC-DELTA Notice that the following rule extends the delta rule defined in section 4.3

$$\frac{E[\Gamma] \vdash p : \mathsf{Struct}\ e_1; \ldots; e_i; \mathsf{Def}()(c := t : U); \ldots\ \mathsf{End}}{E[\Gamma] \vdash p.c \triangleright_\delta t}$$

In the rules below we assume 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 $[c_1 : C_1; \ldots; c_n : C_n]$

ACC-IND

$$\frac{E[\Gamma] \vdash p : \mathsf{Struct}\ e_1; \ldots; e_i; \mathsf{Ind}()[\Gamma_P](\Gamma_C := \Gamma_I); \ldots\ \mathsf{End}}{E[\Gamma] \vdash p.I_j : (p_1 : P_1) \ldots (p_r : P_r) A_j}$$

$$\frac{E[\Gamma] \vdash p : \mathsf{Struct}\ e_1; \ldots; e_i; \mathsf{Ind}()[\Gamma_P](\Gamma_C := \Gamma_I); \ldots\ \mathsf{End}}{E[\Gamma] \vdash p.c_m : (p_1 : P_1) \ldots (p_r : P_r) C_m I_j (I_j\ p_1 \ldots p_r)_{j=1\ldots k}}$$

ACC-INDP

$$\frac{E[] \vdash p : \mathsf{Struct}\ e_1; \ldots; e_i; \mathsf{Ind}_{p'}()[\Gamma_P](\Gamma_C := \Gamma_I)\ ; \ldots\ \mathsf{End}}{E[] \vdash p.I_i \triangleright_\delta p'.I_i}$$

$$\frac{E[] \vdash p : \mathsf{Struct}\ e_1; \ldots; e_i; \mathsf{Ind}_{p'}()[\Gamma_P](\Gamma_C := \Gamma_I)\ ; \ldots\ \mathsf{End}}{E[] \vdash p.c_i \triangleright_\delta p'.c_i}$$

Part II

The proof engine

Chapter 6

Vernacular commands

6.1 Displaying

6.1.1 `Print` *qualid* .

This command displays on the screen informations about the declared or defined object referred by *qualid*.

Error messages:

1. *qualid* `not a defined object`

Variants:

1. `Print Term` *qualid* .
   This is a synonym to `Print` *qualid* when *qualid* denotes a global constant.

2. `About` *qualid* .
   This displays various informations about the object denoted by *qualid*: its kind (module, constant, assumption, inductive, constructor, abbreviation...), long name, type, implicit arguments and argument scopes. It does not print the body of definitions or proofs.

6.1.2 `Print All.`

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

Variants:

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

2. `Print Section` *ident* .
   should correspond to a currently open section, this command displays the objects defined since the beginning of this section.

6.2 Requests to the environment

6.2.1 Check *term* .

This command displays the type of *term*. When called in proof mode, the term is checked in the local context of the current subgoal.

6.2.2 Eval *convtactic* in *term* .

This command performs the specified reduction on *term*, and displays the resulting term with its type. The term to be reduced may depend on hypothesis introduced in the first subgoal (if a proof is in progress).

See also: Section 8.5.

6.2.3 Extraction *term* .

This command displays the extracted term from *term*. The extraction is processed according to the distinction between Set and Prop; that is to say, between logical and computational content (see Section 4.1.1). The extracted term is displayed in Objective Caml syntax, where global identifiers are still displayed as in Coq terms.

Variants:

1. Recursive Extraction $qualid_1$ ... $qualid_n$.
   Recursively extracts all the material needed for the extraction of globals $qualid_1$ ... $qualid_n$.

See also: Chapter 21.

6.2.4 Print Assumptions *qualid* .

This commands display all the assumptions (axioms, parameters and variables) a theorem or definition depends on. Especially, it informs on the assumptions with respect to which the validity of a theorem relies.

6.2.5 Search *qualid* .

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

1. The reference *qualid* was not found in the current environment
   There is no constant in the environment named *qualid*.

Variants:

1. Search *qualid* inside $module_1$ ... $module_n$.
   This restricts the search to constructions defined in modules $module_1$ ... $module_n$.

2. `Search` *qualid* `outside` $module_1$ ... $module_n$ **.**

   This restricts the search to constructions not defined in modules $module_1$ ... $module_n$.

   Error messages:

   (a) `Module/section` *module* `not found` No module *module* has been required (see Section 6.4.1).

6.2.6 `SearchAbout` *qualid* **.**

This command displays the name and type of all objects (theorems, axioms, etc) of the current context whose statement contains *qualid*. This command is useful to remind the user of the name of library lemmas.

Error messages:

1. `The reference` *qualid* `was not found in the current environment`
   There is no constant in the environment named *qualid*.

Variants:

1. `SearchAbout` *string* **.**

   If *string* is a valid identifier, this command displays the name and type of all objects (theorems, axioms, etc) of the current context whose name contains *string*. If *string* is a notation's string denoting some reference *qualid* (referred to by its main symbol as in `"+"` or by its notation's string as in `"_ + _"` or `"_ 'U' _"`, see Section 12.1), the command works like `SearchAbout` *qualid*.

2. `SearchAbout` *string*`%`*key* **.**

   The string *string* must be a notation or the main symbol of a notation which is then interpreted in the scope bound to the delimiting key *key* (see Section 12.2.2).

3. `SearchAbout` *term_pattern* **.**

   This searches for all statements or types of definition that contains a subterm that matches the pattern *term_pattern* (holes of the pattern are either denoted by "_" or by "`?`*ident*" when non linear patterns are expected).

4. `SearchAbout [` *[−]term_pattern-string* ... *[−]term_pattern-string* `]` **.**

   where *term_pattern-string* is a *term_pattern* or a *string*, or a *string* followed by a scope delimiting key `%`*key*.

   This generalization of `SearchAbout` searches for all objects whose statement or type contains a subterm matching *term_pattern* (or *qualid* if *string* is the notation for a reference *qualid*) and whose name contains all *string* of the request that correspond to valid identifiers. If a *term_pattern* or a *string* is prefixed by "-", the search excludes the objects that mention that *term_pattern* or that *string*.

5. `SearchAbout` *term_pattern-string* `inside` $module_1$ ... $module_n$ **.**
   `SearchAbout [` *term_pattern-string* ... *term_pattern-string* `] inside` $module_1$ ... $module_n$
   This restricts the search to constructions defined in modules $module_1$ ... $module_n$.

6. `SearchAbout` *term_pattern-string* `outside` $module_1 \ldots module_n$.
   `SearchAbout [` *term_pattern-string* `...` *term_pattern-string* `] outside` $module_1 \ldots module_n$.
   This restricts the search to constructions not defined in modules $module_1 \ldots module_n$.

Examples:

```
Coq < Require Import ZArith.

Coq < SearchAbout [ Zmult Zplus "distr" ].
weak_Zmult_plus_distr_r:
  forall (p : positive) (n m : Z),
  (Zpos p * (n + m))%Z = (Zpos p * n + Zpos p * m)%Z
Zmult_plus_distr_r:
  forall n m p : Z, (n * (m + p))%Z = (n * m + n * p)%Z
Zmult_plus_distr_l:
  forall n m p : Z, ((n + m) * p)%Z = (n * p + m * p)%Z
fast_Zmult_plus_distr_l:
  forall (n m p : Z) (P : Z -> Prop),
  P (n * p + m * p)%Z -> P ((n + m) * p)%Z

Coq < SearchAbout [ "+"%Z "*"%Z "distr" -positive -Prop].
Zmult_plus_distr_r:
  forall n m p : Z, (n * (m + p))%Z = (n * m + n * p)%Z
Zmult_plus_distr_l:
  forall n m p : Z, ((n + m) * p)%Z = (n * p + m * p)%Z

Coq < SearchAbout (?x * _ + ?x * _)%Z outside OmegaLemmas.
weak_Zmult_plus_distr_r:
  forall (p : positive) (n m : Z),
  (Zpos p * (n + m))%Z = (Zpos p * n + Zpos p * m)%Z
Zmult_plus_distr_r:
  forall n m p : Z, (n * (m + p))%Z = (n * m + n * p)%Z
```

6.2.7 `SearchPattern` *term*.

This command displays the name and type of all theorems of the current context whose statement's conclusion matches the expression *term* where holes in the latter are denoted by "_". It is a variant of `SearchAbout` *term_pattern* that does not look for subterms but searches for statements whose conclusion has exactly the expected form.

```
Coq < Require Import Arith.

Coq < SearchPattern (_ + _ = _ + _).
plus_comm: forall n m : nat, n + m = m + n
plus_Snm_nSm: forall n m : nat, S n + m = n + S m
plus_assoc: forall n m p : nat, n + (m + p) = n + m + p
plus_permute: forall n m p : nat, n + (m + p) = m + (n + p)
plus_assoc_reverse: forall n m p : nat, n + m + p = n + (m + p)
plus_permute_2_in_4:
  forall n m p q : nat, n + m + (p + q) = n + p + (m + q)
```

Patterns need not be linear: you can express that the same expression must occur in two places by using pattern variables '`?`*ident*".

```
Coq < Require Import Arith.
Coq < SearchPattern (?X1 + _ = _ + ?X1).
plus_comm: forall n m : nat, n + m = m + n
```

Variants:

1. `SearchPattern` *term* `inside` $module_1$ `...` $module_n$`.`

   This restricts the search to constructions defined in modules $module_1 \ldots module_n$.

2. `SearchPattern` *term* `outside` $module_1$ `...` $module_n$`.`

   This restricts the search to constructions not defined in modules $module_1 \ldots module_n$.

6.2.8 `SearchRewrite` *term*`.`

This command displays the name and type of all theorems of the current context whose statement's conclusion is an equality of which one side matches the expression *term*. Holes in *term* are denoted by "_".

```
Coq < Require Import Arith.
Coq < SearchRewrite (_ + _ + _).
plus_assoc: forall n m p : nat, n + (m + p) = n + m + p
plus_assoc_reverse: forall n m p : nat, n + m + p = n + (m + p)
plus_permute_2_in_4:
  forall n m p q : nat, n + m + (p + q) = n + p + (m + q)
```

Variants:

1. `SearchRewrite` *term* `inside` $module_1$ `...` $module_n$`.`

   This restricts the search to constructions defined in modules $module_1 \ldots module_n$.

2. `SearchRewrite` *term* `outside` $module_1$ `...` $module_n$`.`

   This restricts the search to constructions not defined in modules $module_1 \ldots module_n$.

6.2.9 `Locate` *qualid*`.`

This command displays the full name of the qualified identifier *qualid* and consequently the COQ module in which it is defined.

```
Coq < Locate nat.
Inductive Coq.Init.Datatypes.nat
Coq < Locate Datatypes.O.
Constructor Coq.Init.Datatypes.O
  (shorter name to refer to it in current context is O)
Coq < Locate Init.Datatypes.O.
Constructor Coq.Init.Datatypes.O
  (shorter name to refer to it in current context is O)
Coq < Locate Coq.Init.Datatypes.O.
Constructor Coq.Init.Datatypes.O
```

```
   (shorter name to refer to it in current context is 0)
Coq < Locate I.Dont.Exist.
No object of suffix I.Dont.Exist
```

See also: Section 12.1.10

6.2.10 The WHELP searching tool

WHELP is an experimental searching and browsing tool for the whole COQ library and the whole set of COQ user contributions. WHELP requires a browser to work. WHELP has been developed at the University of Bologna as part of the HELM[1] and MoWGLI[2] projects. It can be invoked directly from the COQ toplevel or from COQIDE, assuming a graphical environment is also running. The browser to use can be selected by setting the environment variable `COQREMOTEBROWSER`. If not explicitly set, it defaults to `firefox -remote \"OpenURL(%s,new-tab)\" || firefox %s &"` or `C:\\PROGRA~1\\INTERN~1\\IEXPLORE %s`, depending on the underlying operating system (in the command, the string `%s` serves as metavariable for the url to open). The Whelp tool relies on a dedicated Whelp server and on another server called Getter that retrieves formal documents. The default Whelp server name can be obtained using the command `Test Whelp Server` and the default Getter can be obtained using the command: `Test Whelp Getter`. The Whelp server name can be changed using the command:

`Set Whelp Server` *string*.
where *string* is a URL (e.g. `http://mowgli.cs.unibo.it:58080`).

The Getter can be changed using the command:

`Set Whelp Getter` *string*.
where *string* is a URL (e.g. `http://mowgli.cs.unibo.it:58081`).

The WHELP commands are:

`Whelp Locate "`*reg_expr*`".`

This command opens a browser window and displays the result of seeking for all names that match the regular expression *reg_expr* in the COQ library and user contributions. The regular expression can contain the special operators are * and ? that respectively stand for an arbitrary substring and for exactly one character.

Variant: `Whelp Locate` *ident*`.`
This is equivalent to `Whelp Locate "`*ident*`"`.

`Whelp Match` *pattern*`.`

This command opens a browser window and displays the result of seeking for all statements that match the pattern *pattern*. Holes in the pattern are represented by the wildcard character "_".

`Whelp Instance` *pattern*`.`

This command opens a browser window and displays the result of seeking for all statements that are instances of the pattern *pattern*. The pattern is here assumed to be an universally quantified expression.

[1] Hypertextual Electronic Library of Mathematics
[2] Mathematics on the Web, Get it by Logics and Interfaces

`Whelp Elim` *qualid* **.**

This command opens a browser window and displays the result of seeking for all statements that have the "form" of an elimination scheme over the type denoted by *qualid*.

`Whelp Hint` *term* **.**

This command opens a browser window and displays the result of seeking for all statements that can be instantiated so that to prove the statement *term*.

Variant: `Whelp Hint.`
This is equivalent to `Whelp Hint` *goal* where *goal* is the current goal to prove. Notice that Coq does not send the local environment of definitions to the WHELP tool so that it only works on requests strictly based on, only, definitions of the standard library and user contributions.

6.3 Loading files

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`.

6.3.1 `Load` *ident* **.**

This command loads the file named *ident* `.v`, searching successively in each of the directories specified in the *loadpath*. (see Section 6.5)

Variants:

1. `Load` *string* **.**
   Loads the file denoted by the string *string*, where *string* is any complete filename. 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 6.8.1

Error messages:

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

6.4 Compiled files

This section describes the commands used to load compiled files (see Chapter 13 for documentation on how to compile a file). A compiled file is a particular case of module called *library file*.

6.4.1 `Require` *qualid* .

This command looks in the loadpath for a file containing module *qualid* and adds the corresponding module to the environment of Coq. As library files have dependencies in other library files, the command `Require` *qualid* recursively requires all library files the module *qualid* depends on and adds the corresponding modules to the environment of Coq too. Coq assumes that the compiled files have been produced by a valid Coq compiler and their contents are then not replayed nor rechecked.

To locate the file in the file system, *qualid* is decomposed under the form *dirpath* . *ident* and the file *ident* . `vo` is searched in the physical directory of the file system that is mapped in Coq loadpath to the logical path *dirpath* (see Section 6.5). The mapping between physical directories and logical names at the time of requiring the file must be consistent with the mapping used to compile the file.

Variants:

1. `Require Import` *qualid* .

   This loads and declares the module *qualid* and its dependencies then imports the contents of *qualid* as described in Section 2.5.8.

   It does not import the modules on which *qualid* depends unless these modules were itself required in module *qualid* using `Require Export`, as described below, or recursively required through a sequence of `Require Export`.

   If the module required has already been loaded, `Require Import` *qualid* simply imports it, as `Import` *qualid* would.

2. `Require Export` *qualid* .

   This command acts as `Require Import` *qualid*, but if a further module, say *A*, contains a command `Require Export` *B*, then the command `Require Import` *A* also imports the module *B*.

3. `Require` *[*`Import` *|* `Export`*]* $qualid_1$ `...` $qualid_n$.

   This loads the modules $qualid_1$, ..., $qualid_n$ and their recursive dependencies. If `Import` or `Export` is given, it also imports $qualid_1$, ..., $qualid_n$ and all the recursive dependencies that were marked or transitively marked as `Export`.

4. `Require` *[*`Import` *|* `Export`*]* *string* .

   This shortcuts the resolution of the qualified name into a library file name by directly requiring the module to be found in file *string*.vo.

Error messages:

1. `Cannot load` *qualid* `:  no physical path bound to` *dirpath*

2. `Cannot find library foo in loadpath`

   The command did not find the file `foo.vo`. Either `foo.v` exists but is not compiled or `foo.vo` is in a directory which is not in your `LoadPath` (see Section 6.5).

3. `Compiled library` *ident* `.vo makes inconsistent assumptions over library` *qualid*

   The command tried to load library file *ident*.vo that depends on some specific version of library *qualid* which is not the one already loaded in the current Coq session. Probably *ident*.v was not properly recompiled with the last version of the file containing module *qualid*.

4. `Bad magic number`

   The file *ident*`.vo` was found but either it is not a Coq compiled module, or it was compiled with an older and incompatible version of Coq.

5. `The file` *ident*`.vo contains library` *dirpath* `and not library` *dirpath'*

   The library file *dirpath'* is indirectly required by the `Require` command but it is bound in the current loadpath to the file *ident*.vo which was bound to a different library name *dirpath* at the time it was compiled.

See also: Chapter 13

6.4.2 `Print Libraries.`

This command displays the list of library files loaded in the current Coq session. For each of these libraries, it also tells if it is imported.

6.4.3 `Declare ML Module` *string*$_1$ `..` *string*$_n$`.`

This commands loads the Objective Caml compiled files *string*$_1$ ... *string*$_n$ (dynamic link). It is mainly used to load tactics dynamically. The files are searched into the current Objective Caml loadpath (see the command `Add ML Path` in the Section 6.5). Loading of Objective Caml files is only possible under the bytecode version of `coqtop` (i.e. `coqtop` called with options `-byte`, see chapter 13), or when Coq has been compiled with a version of Objective Caml that supports native `Dynlink` ($\geq$ 3.11).

Error messages:

1. `File not found on loadpath :` *string*
2. `Loading of ML object file forbidden in a native Coq`

6.4.4 `Print ML Modules.`

This print the name of all Objective Caml modules loaded with `Declare ML Module`. To know from where these module were loaded, the user should use the command `Locate File` (see Section 6.5.10)

6.5 Loadpath

There are currently two loadpaths in Coq. A loadpath where seeking Coq files (extensions `.v` or `.vo` or `.vi`) and one where seeking Objective Caml files. The default loadpath contains the directory "`.`" denoting the current directory and mapped to the empty logical path (see Section 2.6.2).

6.5.1 `Pwd.`

This command displays the current working directory.

6.5.2 `Cd` *string* .

This command changes the current directory according to *string* which can be any valid path.

Variants:

1. `Cd.`
   Is equivalent to `Pwd.`

6.5.3 `Add LoadPath` *string* `as` *dirpath* .

This command adds the physical directory *string* to the current COQ loadpath and maps it to the logical directory *dirpath*, which means that every file *dirname*/*basename.v* physically lying in subdirectory *string*/*dirname* becomes accessible in COQ through absolute logical name *dirpath* . *dirname* . *basename*.

Remark: `Add LoadPath` also adds *string* to the current ML loadpath.

Variants:

1. `Add LoadPath` *string* .
   Performs as `Add LoadPath` *string* `as` *dirpath* but for the empty directory path.

6.5.4 `Add Rec LoadPath` *string* `as` *dirpath* .

This command adds the physical directory *string* and all its subdirectories to the current COQ loadpath. The top directory *string* is mapped to the logical directory *dirpath* and any subdirectory *pdir* of it is mapped to logical name *dirpath* . *pdir* and recursively. Subdirectories corresponding to invalid COQ identifiers are skipped, and, by convention, subdirectories named `CVS` or `_darcs` are skipped too.

Otherwise, said, `Add Rec LoadPath` *string* `as` *dirpath* behaves as `Add LoadPath` *string* `as` *dirpath* excepts that files lying in validly named subdirectories of *string* need not be qualified to be found.

In case of files with identical base name, files lying in most recently declared *dirpath* are found first and explicit qualification is required to refer to the other files of same base name.

If several files with identical base name are present in different subdirectories of a recursive loadpath declared via a single instance of `Add Rec LoadPath`, which of these files is found first is system-dependent and explicit qualification is recommended.

Remark: `Add Rec LoadPath` also recursively adds *string* to the current ML loadpath.

Variants:

1. `Add Rec LoadPath` *string* .
   Works as `Add Rec LoadPath` *string* `as` *dirpath* but for the empty logical directory path.

6.5.5 `Remove LoadPath` *string* .

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

6.5.6 `Print LoadPath.`

This command displays the current Coq loadpath.

Variants:

1. `Print LoadPath` *dirpath* `.`
   Works as `Print LoadPath` but displays only the paths that extend the *dirpath* prefix.

6.5.7 `Add ML Path` *string* `.`

This command adds the path *string* to the current Objective Caml loadpath (see the command `Declare ML Module` in the Section 6.4).

Remark: This command is implied by `Add LoadPath` *string* `as` *dirpath*.

6.5.8 `Add Rec ML Path` *string* `.`

This command adds the directory *string* and all its subdirectories to the current Objective Caml loadpath (see the command `Declare ML Module` in the Section 6.4).

Remark: This command is implied by `Add Rec LoadPath` *string* `as` *dirpath*.

6.5.9 `Print ML Path` *string* `.`

This command displays the current Objective Caml loadpath. This command makes sense only under the bytecode version of `coqtop`, i.e. using option `-byte` (see the command `Declare ML Module` in the section 6.4).

6.5.10 `Locate File` *string* `.`

This command displays the location of file *string* in the current loadpath. Typically, *string* is a `.cmo` or `.vo` or `.v` file.

6.5.11 `Locate Library` *dirpath* `.`

This command gives the status of the Coq module *dirpath*. It tells if the module is loaded and if not searches in the load path for a module of logical name *dirpath*.

6.6 States and Reset

6.6.1 `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 messages:

1. *ident*: `no such entry`

6.6.2 `Back.`

This commands undoes all the effects of the last vernacular command. This does not include commands that only access to the environment like those described in the previous sections of this chapter (for instance `Require` and `Load` can be undone, but not `Check` and `Locate`). Commands read from a vernacular file are considered as a single command.

Variants:

1. `Back` n
   Undoes n vernacular commands.

Error messages:

1. `Reached begin of command history`
   Happens when there is vernacular command to undo.

6.6.3 `Backtrack` num_1 num_2 num_3`.`

This command is dedicated for the use in graphical interfaces. It allows to backtrack to a particular *global* state, i.e. typically a state corresponding to a previous line in a script. A global state includes declaration environment but also proof environment (see Chapter 7). The three numbers num_1, num_2 and num_3 represent the following:

- num_3: Number of `Abort` to perform, i.e. the number of currently opened nested proofs that must be canceled (see Chapter 7).
- num_2: *Proof state number* to unbury once aborts have been done. Coq will compute the number of `Undo` to perform (see Chapter 7).
- num_1: Environment state number to unbury, Coq will compute the number of `Back` to perform.

How to get state numbers?

Notice that when in `-emacs` mode, Coq displays the current proof and environment state numbers in the prompt. More precisely the prompt in `-emacs` mode is the following:
`<prompt>` id_i < num_1 | id_1 | id_2 | ... | id_n | num_2 < `</prompt>`
Where:

- id_i is the name of the current proof (if there is one, otherwise `Coq` is displayed, see Chapter 7).
- num_1 is the environment state number after the last command.
- num_2 is the proof state number after the last command.
- id_1 id_2 ... id_n are the currently opened proof names (order not significant).

It is then possible to compute the `Backtrack` command to unbury the state corresponding to a particular prompt. For example, suppose the current prompt is:
< goal4 < 35 | goal1 | goal4 | goal3 | goal2 | | 8 < `</prompt>`
and we want to backtrack to a state labeled by:
< goal2 < 32 | goal1 | goal2 | 12 < `</prompt>`

We have to perform `Backtrack 32 12 2` , i.e. perform 2 `Aborts` (to cancel goal4 and goal3), then rewind proof until state 12 and finally go back to environment state 32. Notice that this supposes that proofs are nested in a regular way (no `Resume` or `Suspend` commands).

Variants:

1. `BackTo n.`
   Is a more basic form of `Backtrack` where only the first argument (global environment number) is given, no `abort` and no `Undo` is performed.

6.6.4 `Restore State` *string* `.`

Restores the state contained in the file *string*.

Variants:

1. `Restore State` *ident*
   Equivalent to `Restore State "`*ident*`.coq"`.

2. `Reset Initial.`
   Goes back to the initial state (like after the command `coqtop`, when the interactive session began). This command is only available interactively.

6.6.5 `Write State` *string* `.`

Writes the current state into a file *string* for use in a further session. This file can be given as the `inputstate` argument of the commands `coqtop` and `coqc`.

Variants:

1. `Write State` *ident*
   Equivalent to `Write State "`*ident*`.coq"`. The state is saved in the current directory (see Section 6.5.1).

6.7 Quitting and debugging

6.7.1 `Quit.`

This command permits to quit Coq.

6.7.2 `Drop.`

This is used mostly as a debug facility by Coq's implementors and does not concern the casual user. This command permits to leave Coq temporarily and enter the Objective Caml toplevel. The Objective Caml command:

```
#use "include";;
```

add the right loadpaths and loads some toplevel printers for all abstract types of Coq- section_path, identifiers, terms, judgments, ... You can also use the file `base_include` instead, that loads only the pretty-printers for section_paths and identifiers. You can return back to Coq with the command:

```
go();;
```

Warnings:

1. It only works with the bytecode version of Coq (i.e. `coqtop` called with option `-byte`, see the contents of Section 13.1).
2. You must have compiled Coq from the source package and set the environment variable `COQTOP` to the root of your copy of the sources (see Section 13.4).

6.7.3 `Time` *command* .

This command executes the vernacular command *command* and display the time needed to execute it.

6.8 Controlling display

6.8.1 `Set Silent.`

This command turns off the normal displaying.

6.8.2 `Unset Silent.`

This command turns the normal display on.

6.8.3 `Set Printing Width` *integer* .

This command sets which left-aligned part of the width of the screen is used for display.

6.8.4 `Unset Printing Width.`

This command resets the width of the screen used for display to its default value (which is 78 at the time of writing this documentation).

6.8.5 `Test Printing Width.`

This command displays the current screen width used for display.

6.8.6 `Set Printing Depth` *integer* .

This command sets the nesting depth of the formatter used for pretty-printing. Beyond this depth, display of subterms is replaced by dots.

6.8.7 `Unset Printing Depth.`

This command resets the nesting depth of the formatter used for pretty-printing to its default value (at the time of writing this documentation, the default value is 50).

6.8.8 `Test Printing Depth.`

This command displays the current nesting depth used for display.

6.9 Controlling the reduction strategies and the conversion algorithm

Coq provides reduction strategies that the tactics can invoke and two different algorithms to check the convertibility of types. The first conversion algorithm lazily compares applicative terms while the other is a brute-force but efficient algorithm that first normalizes the terms before comparing them. The second algorithm is based on a bytecode representation of terms similar to the bytecode representation used in the ZINC virtual machine [90]. It is specially useful for intensive computation of algebraic values, such as numbers, and for reflexion-based tactics. The commands to fine-tune the reduction strategies and the lazy conversion algorithm are described first.

6.9.1 `Opaque` *qualid*$_1$ ... *qualid*$_n$.

This command has an effect on unfoldable constants, i.e. on constants defined by `Definition` or `Let` (with an explicit body), or by a command assimilated to a definition such as `Fixpoint`, `Program Definition`, etc, or by a proof ended by `Defined`. The command tells not to unfold the constants *qualid*$_1$ ... *qualid*$_n$ in tactics using δ-conversion (unfolding a constant is replacing it by its definition).

`Opaque` has also on effect on the conversion algorithm of Coq, telling to delay the unfolding of a constant as later as possible in case Coq has to check the conversion (see Section 4.3) of two distinct applied constants.

The scope of `Opaque` is limited to the current section, or current file, unless the variant `Global Opaque` *qualid*$_1$ ... *qualid*$_n$ is used.

See also: sections 8.5, 8.12, 7.1.4

Error messages:

1. `The reference` *qualid* `was not found in the current environment`
   There is no constant referred by *qualid* in the environment. Nevertheless, if you asked `Opaque foo bar` and if `bar` does not exist, `foo` is set opaque.

6.9.2 `Transparent` *qualid*$_1$ ... *qualid*$_n$.

This command is the converse of `Opaque` and it applies on unfoldable constants to restore their unfoldability after an `Opaque` command.

Note in particular that constants defined by a proof ended by `Qed` are not unfoldable and `Transparent` has no effect on them. 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.

The scope of `Transparent` is limited to the current section, or current file, unless the variant `Global Transparent` *qualid*$_1$ ... *qualid*$_n$ is used.

Error messages:

1. `The reference` *qualid* `was not found in the current environment`
   There is no constant referred by *qualid* in the environment.

See also: sections 8.5, 8.12, 7.1.4

6.9.3 `Strategy` *level* `[` $qualid_1$ `...` $qualid_n$ `].`

This command generalizes the behavior of `Opaque` and `Transparent` commands. It is used to fine-tune the strategy for unfolding constants, both at the tactic level and at the kernel level. This command associates a level to $qualid_1 \ldots qualid_n$. Whenever two expressions with two distinct head constants are compared (for instance, this comparison can be triggered by a type cast), the one with lower level is expanded first. In case of a tie, the second one (appearing in the cast type) is expanded.

Levels can be one of the following (higher to lower):

opaque : level of opaque constants. They cannot be expanded by tactics (behaves like $+\infty$, see next item).

num : levels indexed by an integer. Level 0 corresponds to the default behavior, which corresponds to transparent constants. This level can also be referred to as **transparent**. Negative levels correspond to constants to be expanded before normal transparent constants, while positive levels correspond to constants to be expanded after normal transparent constants.

expand : level of constants that should be expanded first (behaves like $-\infty$)

These directives survive section and module closure, unless the command is prefixed by `Local`. In the latter case, the behavior regarding sections and modules is the same as for the `Transparent` and `Opaque` commands.

6.9.4 `Set Virtual Machine`

This activates the bytecode-based conversion algorithm.

6.9.5 `Unset Virtual Machine`

This deactivates the bytecode-based conversion algorithm.

6.9.6 `Test Virtual Machine`

This tells if the bytecode-based conversion algorithm is activated. The default behavior is to have the bytecode-based conversion algorithm deactivated.

See also: sections 8.5.1 and 13.5.

Chapter 7

Proof handling

In Coq's proof editing mode all top-level commands documented in Chapter 6 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 8.
When switching in editing proof mode, the prompt `Coq <` is changed into *ident* `<` where *ident* is the declared name of the theorem currently edited.

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.

To each subgoal is associated a number of 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 8.3.5).

When a proof is achieved the message `Proof completed` 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* [75, 6, 71, 78], Coq stores proofs as terms of Cic. Those terms are called *proof terms*.

It is possible to edit several proofs at the same time: see section 7.1.8

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`.

7.1 Switching on/off the proof editing mode

7.1.1 `Goal` *form* .

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

Error messages:

1. `the term` *form* `has type ... which should be Set, Prop or Type`

See also: Section 7.1.4

7.1.2 `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 top-level 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 messages:

1. `Attempt to save an incomplete proof`
2. Sometimes an error occurs when building the proof term, because tactics do not enforce completely the term construction constraints.

   The user should also be aware of the fact that since the proof term is completely rechecked at this point, one may have to wait a while when the proof is large. In some exceptional cases one may even incur a memory overflow.

Variants:

1. `Defined.`

   Defines the proved term as a transparent constant.

2. `Save.`

   Is equivalent to `Qed`.

3. `Save` *ident* `.`

   Forces the name of the original goal to be *ident*. This command (and the following ones) can only be used if the original goal has been opened using the `Goal` command.

4. `Save Theorem` *ident* `.`
   `Save Lemma` *ident* `.`
   `Save Remark` *ident* `.`
   `Save Fact` *ident* `.`

   Are equivalent to `Save` *ident* `.`

7.1.3 `Admitted.`

This command is available in interactive editing proof mode to give up the current proof and declare the initial goal as an axiom.

7.1.4 `Theorem` *ident* `:` *form* `.`

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

Error messages:

1. `the term` *form* `has type ... which should be Set, Prop or Type`

2. *ident*`already exists`

   The name you provided already defined. You have then to choose another name.

Variants:

1. `Lemma` *ident* `:` *form* `.`

   It is equivalent to `Theorem` *ident* `:` *form* `.`

2. `Remark` *ident* `:` *form* `.`
   `Fact` *ident* `:` *form* `.`

   Used to have a different meaning, but are now equivalent to `Theorem` *ident* `:` *form* `.` They are kept for compatibility.

3. `Definition` *ident* `:` *form* `.`

   Analogous to `Theorem`, intended to be used in conjunction with `Defined` (see 1) in order to define a transparent constant.

4. `Let` *ident* `:` *form* `.`

   Analogous to `Definition` except that the definition is turned into a local definition on objects depending on it after closing the current section.

7.1.5 `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 8.2.1).

Variant: `Proof.`

Is a noop which is useful to delimit the sequence of tactic commands which start a proof, after a `Theorem` command. It is a good practice to use `Proof.` as an opening parenthesis, closed in the script with a closing `Qed.`

See also: `Proof with` *tactic* `.` in Section 8.13.6.

7.1.6 `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 messages:

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.

7.1.7 `Suspend.`

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

7.1.8 `Resume.`

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

Error messages:

1. `No proof-editing in progress`

Variants:

1. `Resume` *ident* `.`

   Restarts the editing of the proof named *ident*. This can be used to navigate between currently edited proofs.

Error messages:

1. `No such proof`

7.1.9 `Existential` *num* `:=` *term* `.`

This command allows to instantiate an existential variable. *num* is an index in the list of uninstantiated existential variables displayed by `Show Existentials.` (described in Section 7.3.1)

This command is intented to be used to instantiate existential variables when the proof is completed but some uninstantiated existential variables remain. To instantiate existential variables during proof edition, you should use the tactic `instantiate`.

See also: `instantiate (`*num*`:=` *term*`).` in Section 8.3.15.

7.2 Navigation in the proof tree

7.2.1 `Undo.`

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

Error messages:

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

Variants:

1. `Undo` *num* `.`

   Repeats `Undo` *num* times.

7.2.2 `Set Undo` *num*`.`

This command changes the maximum number of `Undo`'s that will be possible when doing a proof. It only affects proofs started after this command, such that if you want to change the current undo limit inside a proof, you should first restart this proof.

7.2.3 `Unset Undo.`

This command resets the default number of possible `Undo` commands (which is currently 12).

7.2.4 `Restart.`

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

Error messages:

1. `No focused proof to restart`

7.2.5 `Focus.`

This focuses the attention on the first subgoal to prove and the printing of the other subgoals is suspended until the focused subgoal is solved or unfocused. This is useful when there are many current subgoals which clutter your screen.

Variant:

1. `Focus` *num*`.`
   This focuses the attention on the num^{th} subgoal to prove.

7.2.6 `Unfocus.`

Turns off the focus mode.

7.3 Requesting information

7.3.1 `Show.`

This command displays the current goals.

Variants:

1. `Show` *num*`.`
   Displays only the *num*-th subgoal.
   Error messages:
   (a) `No such goal`
   (b) `No focused proof`
2. `Show Implicits.`
   Displays the current goals, printing the implicit arguments of constants.

3. `Show Implicits` *num*`.`
   Same as above, only displaying the *num*-th subgoal.

4. `Show Script.`
   Displays the whole list of tactics applied from the beginning of the current proof. This tactics script may contain some holes (subgoals not yet proved). They are printed under the form `<Your Tactic Text here>`.

5. `Show Tree.`
   This command can be seen as a more structured way of displaying the state of the proof than that provided by `Show Script`. Instead of just giving the list of tactics that have been applied, it shows the derivation tree constructed by then. Each node of the tree contains the conclusion of the corresponding sub-derivation (i.e. a goal with its corresponding local context) and the tactic that has generated all the sub-derivations. The leaves of this tree are the goals which still remain to be proved.

6. `Show Proof.`
   It displays the proof term generated by the tactics that have been applied. If the proof is not completed, this term contain holes, which correspond to the sub-terms which are still to be constructed. These holes appear as a question mark indexed by an integer, and applied to the list of variables in the context, since it may depend on them. The types obtained by abstracting away the context from the type of each hole-placer are also printed.

7. `Show Conjectures.`
   It prints the list of the names of all the theorems that are currently being proved. As it is possible to start proving a previous lemma during the proof of a theorem, this list may contain several names.

8. `Show Intro.`
   If the current goal begins by at least one product, this command prints the name of the first product, as it would be generated by an anonymous `Intro`. The aim of this command is to ease the writing of more robust scripts. For example, with an appropriate Proof General macro, it is possible to transform any anonymous `Intro` into a qualified one such as `Intro y13`. In the case of a non-product goal, it prints nothing.

9. `Show Intros.`
   This command is similar to the previous one, it simulates the naming process of an `Intros`.

10. `Show Existentials`
    It displays the set of all uninstantiated existential variables in the current proof tree, along with the type and the context of each variable.

7.3.2 `Guarded.`

Some tactics (e.g. refine 8.2.2) allow to build proofs using fixpoint or co-fixpoint constructions. Due to the incremental nature of interactive proof construction, the check of the termination (or guardedness) of the recursive calls in the fixpoint or cofixpoint constructions is postponed to the time of the completion of the proof.

The command `Guarded` allows to verify if the guard condition for fixpoint and cofixpoint is violated at some time of the construction of the proof without having to wait the completion of the proof."

7.3.3 `Set Hyps Limit` *num*`.`

This command sets the maximum number of hypotheses displayed in goals after the application of a tactic. All the hypotheses remains usable in the proof development.

7.3.4 `Unset Hyps Limit.`

This command goes back to the default mode which is to print all available hypotheses.

Chapter 8

Tactics

A deduction rule is a link between some (unique) formula, that we call the *conclusion* and (several) formulas that we call the *premises*. 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 premises to conclusion. The other way says: *"to prove this I have to prove this and this"*. 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 premises. We say that the conclusion is *the goal* to prove and premises 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:tactic* which means *"apply tactic tactic to goal number n"*. We can show the list of subgoals by typing `Show` (see Section 7.3.1).

Since not every rule applies to a given 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.

Tactics are build from atomic tactics and tactic expressions (which extends the folklore notion of tactical) to combine those atomic tactics. This chapter is devoted to atomic tactics. The tactic language will be described in Chapter 9.

There are, at least, three levels of atomic tactics. The simplest one implements basic rules of the logical framework. The second level is the one of *derived rules* which are built by combination of other tactics. The third one implements heuristics or decision procedures to build a complete proof of a goal.

8.1 Invocation of tactics

A tactic is applied as an ordinary command. If the tactic does not address the first subgoal, the command may be preceded by the wished subgoal number as shown below:

tactic_invocation ::= *num* **:** *tactic* **.**
| *tactic* **.**

8.2 Explicit proof as a term

8.2.1 `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 (see Section 4.3).

Error messages:

1. `Not an exact proof`

Variants:

1. `eexact` *term*

   This tactic behaves like `exact` but is able to handle terms with meta-variables.

8.2.2 `refine` *term*

This tactic allows to give an exact proof but still with some holes. The holes are noted "_".

Error messages:

1. `invalid argument`: the tactic `refine` doesn't know what to do with the term you gave.
2. `Refine passed ill-formed term`: the term you gave is not a valid proof (not easy to debug in general). This message may also occur in higher-level tactics, which call `refine` internally.
3. `Cannot infer a term for this placeholder` there is a hole in the term you gave which type cannot be inferred. Put a cast around it.

An example of use is given in Section 10.1.

8.3 Basics

Tactics presented in this section implement the basic typing rules of pCIC given in Chapter 4.

8.3.1 `assumption`

This tactic applies to any goal. It implements the "Var" rule given in Section 4.2. It looks in the local context for an hypothesis which type is equal to the goal. If it is the case, the subgoal is proved. Otherwise, it fails.

Error messages:

1. `No such assumption`

Variants:

1. `eassumption`

   This tactic behaves like `assumption` but is able to handle goals with meta-variables.

8.3.2 clear *ident*

This tactic 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.

Variants:

1. clear $ident_1$ ... $ident_n$

   This is equivalent to clear $ident_1$. ... clear $ident_n$.

2. clearbody *ident*

   This tactic expects *ident* to be a local definition then clears its body. Otherwise said, this tactic turns a definition into an assumption.

3. clear - $ident_1$ ... $ident_n$

   This tactic clears all hypotheses except the ones depending in the hypotheses named $ident_1$ ... $ident_n$ and in the goal.

4. clear

   This tactic clears all hypotheses except the ones depending in goal.

Error messages:

1. *ident* not found
2. *ident* is used in the conclusion
3. *ident* is used in the hypothesis *ident'*

8.3.3 move $ident_1$ after $ident_2$

This moves the hypothesis named $ident_1$ in the local context after the hypothesis named $ident_2$.

If $ident_1$ comes before $ident_2$ in the order of dependences, then all hypotheses between $ident_1$ and $ident_2$ which (possibly indirectly) depend on $ident_1$ are moved also.

If $ident_1$ comes after $ident_2$ in the order of dependences, then all hypotheses between $ident_1$ and $ident_2$ which (possibly indirectly) occur in $ident_1$ are moved also.

Variants:

1. move $ident_1$ before $ident_2$

   This moves $ident_1$ towards and just before the hypothesis named $ident_2$.

2. move *ident* at top

   This moves *ident* at the top of the local context (at the beginning of the context).

3. move *ident* at bottom

   This moves *ident* at the bottom of the local context (at the end of the context).

Error messages:

1. $ident_i$ not found
2. Cannot move $ident_1$ after $ident_2$: it occurs in $ident_2$
3. Cannot move $ident_1$ after $ident_2$: it depends on $ident_2$

8.3.4 `rename` *ident*$_1$ `into` *ident*$_2$

This renames hypothesis *ident*$_1$ into *ident*$_2$ in the current context[1]

Variants:

1. `rename` *ident*$_1$ `into` *ident*$_2$`, ...,` *ident*$_{2k-1}$ `into` *ident*$_{2k}$

   Is equivalent to the sequence of the corresponding atomic `rename`.

Error messages:

1. *ident*$_2$ `not found`
2. *ident*$_2$ `is already used`

8.3.5 `intro`

This tactic applies to a goal which is either a product or starts with a let binder. If the goal is a product, the tactic implements the "Lam" rule given in Section 4.2[2]. If the goal starts with a let binder then the tactic implements a mix of the "Let" and "Conv".

If the current goal is a dependent product `forall` x`:`T`,` U (resp `let` x`:=`t `in` U) then `intro` puts x`:`T (resp x`:=`t) in the local context. The new subgoal is U.

If the goal is a non dependent product T `->` U, then it puts in the local context either `H`n`:`T (if T is of type `Set` or `Prop`) or `X`n`:`T (if the type of T is `Type`). The optional index n is such that `H`n or `X`n is a fresh identifier. In both cases the new subgoal is U.

If the goal is neither a product nor starting with a let definition, the tactic `intro` applies the tactic `red` until the tactic `intro` can be applied or the goal is not reducible.

Error messages:

1. `No product even after head-reduction`
2. *ident* `is already used`

Variants:

1. `intros`

   Repeats `intro` until it meets the head-constant. It never reduces head-constants and it never fails.

2. `intro` *ident*

   Applies `intro` but forces *ident* to be the name of the introduced hypothesis.

   Error message: `name` *ident* `is already used`

   Remark: If a name used by `intro` hides the base name of a global constant then the latter can still be referred to by a qualified name (see 2.6.2).

[1] but it does not rename the hypothesis in the proof-term...

[2] Actually, only the second subgoal will be generated since the other one can be automatically checked.

3. `intros` *ident*$_1$ `...` *ident*$_n$

   Is equivalent to the composed tactic `intro` *ident*$_1$`; ... ; intro` *ident*$_n$.

   More generally, the `intros` tactic takes a pattern as argument in order to introduce names for components of an inductive definition or to clear introduced hypotheses; This is explained in 8.7.3.

4. `intros until` *ident*

   Repeats `intro` until it meets a premise of the goal having form `(` *ident* `:` *term* `)` and discharges the variable named *ident* of the current goal.

   Error message: `No such hypothesis in current goal`

5. `intros until` *num*

   Repeats `intro` until the *num*-th non-dependent product. For instance, on the subgoal `forall x y:nat, x=y -> y=x` the tactic `intros until 1` is equivalent to `intros x y H`, as `x=y -> y=x` is the first non-dependent product. And on the subgoal `forall x y z:nat, x=y -> y=x` the tactic `intros until 1` is equivalent to `intros x y z` as the product on `z` can be rewritten as a non-dependent product: `forall x y:nat, nat -> x=y -> y=x`

   Error message: `No such hypothesis in current goal`

   Happens when *num* is 0 or is greater than the number of non-dependent products of the goal.

6. `intro after` *ident*
   `intro before` *ident*
   `intro at top`
   `intro at bottom`

   Applies `intro` and moves the freshly introduced hypothesis respectively after the hypothesis *ident*, before the hypothesis *ident*, at the top of the local context, or at the bottom of the local context. All hypotheses on which the new hypothesis depends are moved too so as to respect the order of dependencies between hypotheses. Note that `intro at bottom` is a synonym for `intro` with no argument.

   Error messages:

   (a) `No product even after head-reduction`

   (b) `No such hypothesis :` *ident*

7. `intro` *ident*$_1$ `after` *ident*$_2$
   `intro` *ident*$_1$ `before` *ident*$_2$
   `intro` *ident*$_1$ `at top`
   `intro` *ident*$_1$ `at bottom`

   Behaves as previously but naming the introduced hypothesis *ident*$_1$. It is equivalent to `intro` *ident*$_1$ followed by the appropriate call to `move` (see Section 8.3.3).

   Error messages:

   (a) `No product even after head-reduction`

   (b) `No such hypothesis :` *ident*

8.3.6 `apply` *term*

This tactic applies to any goal. The argument *term* is a term well-formed in the local context. The tactic `apply` tries to match the current goal against the conclusion of the type of *term*. If it succeeds, then the tactic returns as many subgoals as the number of non dependent premises of the type of *term*. If the conclusion of the type of *term* does not match the goal *and* the conclusion is an inductive type isomorphic to a tuple type, then each component of the tuple is recursively matched to the goal in the left-to-right order.

The tactic `apply` relies on first-order unification with dependent types unless the conclusion of the type of *term* is of the form $(P\ t_1\ \ldots\ t_n)$ with P to be instantiated. In the latter case, the behavior depends on the form of the goal. If the goal is of the form $(\texttt{fun}\ x\ \texttt{=>}\ Q)\ u_1\ \ldots\ u_n$ and the t_i and u_i unifies, then P is taken to be (fun x => Q). Otherwise, `apply` tries to define P by abstracting over $t_1 \ldots t_n$ in the goal. See `pattern` in Section 8.5.7 to transform the goal so that it gets the form $(\texttt{fun}\ x\ \texttt{=>}\ Q)\ u_1\ \ldots\ u_n$.

Error messages:

1. `Impossible to unify ... with ...`

   The `apply` tactic failed to match the conclusion of *term* and the current goal. You can help the `apply` tactic by transforming your goal with the `change` or `pattern` tactics (see sections 8.5.7, 8.3.11).

2. `Unable to find an instance for the variables` *ident* `...` *ident*

   This occurs when some instantiations of the premises 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 one of the variants below:

Variants:

1. `apply` *term* `with` $\textit{term}_1\ \ldots\ \textit{term}_n$

   Provides `apply` with explicit instantiations for all dependent premises of the type of *term* which do not occur in the conclusion and consequently cannot be found by unification. Notice that $\textit{term}_1 \ldots \textit{term}_n$ must be given according to the order of these dependent premises of the type of *term*.

   Error message: `Not the right number of missing arguments`

2. `apply` *term* `with` $(\textit{ref}_1\ \texttt{:=}\ \textit{term}_1)\ \ldots\ (\textit{ref}_n\ \texttt{:=}\ \textit{term}_n)$

   This also provides `apply` with values for instantiating premises. Here, variables are referred by names and non-dependent products by increasing numbers (see syntax in Section 8.3.17).

3. `apply` $\textit{term}_1$ `,` $\ldots$ `,` $\textit{term}_n$

   This is a shortcut for `apply` $\textit{term}_1$ `; [ .. |` $\ldots$ `; [ .. |` `apply` $\textit{term}_n$ `] ... ]`, i.e. for the successive applications of $\textit{term}_{i+1}$ on the last subgoal generated by `apply` $\textit{term}_i$, starting from the application of $\textit{term}_1$.

4. `eapply` *term*

   The tactic `eapply` behaves as `apply` but does not fail when no instantiation are deducible for some variables in the premises. Rather, it turns these variables into so-called existential variables

which are variables still to instantiate. An existential variable is identified by a name of the form `?`n where n is a number. The instantiation is intended to be found later in the proof.

An example of use of `eapply` is given in Section 10.2.

5. `simple apply` *term*

   This behaves like `apply` but it reasons modulo conversion only on subterms that contain no variables to instantiate. For instance, if `id := fun x:nat => x` and `H : forall y, id y = y` then `simple apply H` on goal `O = O` does not succeed because it would require the conversion of `f ?y` and `O` where `?y` is a variable to instantiate. Tactic `simple apply` does not either traverse tuples as `apply` does.

   Because it reasons modulo a limited amount of conversion, `simple apply` fails quicker than `apply` and it is then well-suited for uses in used-defined tactics that backtrack often.

6. *[*`simple`*]* `apply` *term*$_1$ *[*`with` *bindings_list*$_1$*]* `,` ... `,` *term*$_n$ *[*`with` *bindings_list*$_n$*]*
   [`simple`*]* `eapply` *term*$_1$ *[*`with` *bindings_list*$_1$*]* `,` ... `,` *term*$_n$ *[*`with` *bindings_list*$_n$*]*

   This summarizes the different syntaxes for `apply`.

7. `lapply` *term*

   This tactic applies to any goal, say `G`. The argument *term* has to be well-formed in the current context, its type being reducible to a non-dependent product `A -> B` with `B` possibly containing products. Then it generates two subgoals `B->G` and `A`. Applying `lapply H` (where `H` has type `A->B` and `B` does not start with a product) does the same as giving the sequence `cut B. 2:apply H.` where `cut` is described below.

   Warning: When *term* contains more than one non dependent product the tactic `lapply` only takes into account the first product.

8.3.7 `set` (*ident* `:=` *term*)

This replaces *term* by *ident* in the conclusion or in the hypotheses of the current goal and adds the new definition *ident* `:=` *term* to the local context. The default is to make this replacement only in the conclusion.

Variants:

1. `set` (*ident* `:=` *term*) `in` *goal_occurrences*

   This notation allows to specify which occurrences of *term* have to be substituted in the context. The `in` *goal_occurrences* clause is an occurrence clause whose syntax and behavior is described in Section 8.3.18.

2. `set` (*ident binder* ... *binder* `:=` *term*)

   This is equivalent to `set` (*ident* `:=` `fun` *binder* ... *binder* `=>` *term*).

3. `set` *term*

   This behaves as `set` (*ident* := *term*) but *ident* is generated by Coq. This variant also supports an occurrence clause.

4. `set` (*ident*$_0$ *binder* ... *binder* `:=` *term*) `in` *goal_occurrences*
   `set` *term* `in` *goal_occurrences*

   These are the general forms which combine the previous possibilities.

5. `remember` *term* `as` *ident*

   This behaves as `set` (*ident* := *term*) `in` `*` and using a logical (Leibniz's) equality instead of a local definition.

6. `remember` *term* `as` *ident* `in` *goal_occurrences*

   This is a more general form of `remember` that remembers the occurrences of *term* specified by an occurrences set.

7. `pose` (*ident* `:=` *term*)

   This adds the local definition *ident* := *term* to the current context without performing any replacement in the goal or in the hypotheses. It is equivalent to `set` (*ident* `:=` *term*) `in` `|-`.

8. `pose` (*ident binder* ... *binder* `:=` *term*)

   This is equivalent to `pose` (*ident* `:=` `fun` *binder* ... *binder* => *term*).

9. `pose` *term*

   This behaves as `pose` (*ident* := *term*) but *ident* is generated by Coq.

8.3.8 `assert` (*ident* `:` *form*)

This tactic applies to any goal. `assert (H : U)` adds a new hypothesis of name `H` asserting `U` to the current goal and opens a new subgoal `U`[3]. The subgoal `U` comes first in the list of subgoals remaining to prove.

Error messages:

1. `Not a proposition or a type`

   Arises when the argument *form* is neither of type `Prop`, `Set` nor `Type`.

Variants:

1. `assert` *form*

   This behaves as `assert` (*ident* : *form*) but *ident* is generated by Coq.

2. `assert` (*ident* := *term*)

   This behaves as `assert (`*ident* `:` *type*`); [exact` *term*`|idtac]` where *type* is the type of *term*.

3. `cut` *form*

   This tactic applies to any goal. It implements the non dependent case of the "App" rule given in Section 4.2. (This is Modus Ponens inference rule.) `cut U` transforms the current goal `T` into the two following subgoals: `U -> T` and `U`. The subgoal `U -> T` comes first in the list of remaining subgoal to prove.

[3] This corresponds to the cut rule of sequent calculus.

4. `assert` *form* `by` *tactic*

   This tactic behaves like `assert` but applies *tactic* to solve the subgoals generated by `assert`.

5. `assert` *form* `as` *intro_pattern*

   If *intro_pattern* is a naming introduction pattern (see Section 8.7.3), the hypothesis is named after this introduction pattern (in particular, if *intro_pattern* is *ident*, the tactic behaves like `assert (`*ident* `:` *form*`)`).

   If *intro_pattern* is a disjunctive/conjunctive introduction pattern, the tactic behaves like `assert` *form* then destructing the resulting hypothesis using the given introduction pattern.

6. `assert` *form* `as` *intro_pattern* `by` *tactic*

   This combines the two previous variants of `assert`.

7. `pose proof` *term* `as` *intro_pattern*

   This tactic behaves like `assert T as` *intro_pattern* `by exact` *term* where `T` is the type of *term*.

   In particular, `pose proof` *term* `as` *ident* behaves as `assert (`*ident*`:T) by exact` *term* (where `T` is the type of *term*) and `pose proof` *term* `as` *disj_conj_intro_pattern* behaves like `destruct` *term* `as` *disj_conj_intro_pattern*.

8. `specialize (`*ident* $term_1$ `...` $term_n$`)`
   `specialize` *ident* `with` *bindings_list*

   The tactic `specialize` works on local hypothesis *ident*. The premises of this hypothesis (either universal quantifications or non-dependent implications) are instantiated by concrete terms coming either from arguments $term_1$ ... $term_n$ or from a bindings list (see Section 8.3.17 for more about bindings lists). In the second form, all instantiation elements must be given, whereas in the first form the application to $term_1$ ... $term_n$ can be partial. The first form is equivalent to `assert (`*ident*`':=`*ident* $term_1$ `...` $term_n$`); clear` *ident*`; rename` *ident*`' into` *ident*.

   The name *ident* can also refer to a global lemma or hypothesis. In this case, for compatibility reasons, the behavior of `specialize` is close to that of `generalize`: the instantiated statement becomes an additional premise of the goal.

8.3.9 `apply` *term* `in` *ident*

This tactic applies to any goal. The argument *term* is a term well-formed in the local context and the argument *ident* is an hypothesis of the context. The tactic `apply` *term* `in` *ident* tries to match the conclusion of the type of *ident* against a non dependent premise of the type of *term*, trying them from right to left. If it succeeds, the statement of hypothesis *ident* is replaced by the conclusion of the type of *term*. The tactic also returns as many subgoals as the number of other non dependent premises in the type of *term* and of the non dependent premises of the type of *ident*. If the conclusion of the type of *term* does not match the goal *and* the conclusion is an inductive type isomorphic to a tuple type, then the tuple is (recursively) decomposed and the first component of the tuple of which a non dependent premise matches the conclusion of the type of *ident*. Tuples are decomposed in a width-first left-to-right order (for instance if the type of `H1` is a `A <-> B` statement, and the type of `H2` is `A` then `apply H1`

`in H2` transforms the type of `H2` into `B`). The tactic `apply` relies on first-order pattern-matching with dependent types.

Error messages:

1. `Statement without assumptions`

   This happens if the type of *term* has no non dependent premise.

2. `Unable to apply`

   This happens if the conclusion of *ident* does not match any of the non dependent premises of the type of *term*.

Variants:

1. `apply` *term* `,` `...` `,` *term* `in` *ident*

   This applies each of *term* in sequence in *ident*.

2. `apply` *term bindings_list* `,` `...` `,` *term bindings_list* `in` *ident*

   This does the same but uses the bindings in each *bindings_list* to instantiate the parameters of the corresponding type of *term* (see syntax of bindings in Section 8.3.17).

3. `eapply` *term bindings_list* `,` `...` `,` *term bindings_list* `in` *ident*

   This works as `apply` *term bindings_list* `,` `...` `,` *term bindings_list* `in` *ident* but turns unresolved bindings into existential variables, if any, instead of failing.

4. `apply` *term*`,` *bindings_list* `,` `...` `,` *term*`,` *bindings_list* `in` *ident* `as` *disj_conj_intro_pattern*

   This works as `apply` *term*`,` *bindings_list* `,` `...` `,` *term*`,` *bindings_list* `in` *ident* then destructs the hypothesis *ident* along *disj_conj_intro_pattern* as `destruct` *ident* `as` *disj_conj_intro_pattern* would.

5. `eapply` *term*`,` *bindings_list* `,` `...` `,` *term*`,` *bindings_list* `in` *ident* `as` *disj_conj_intro_pattern*

   This works as `apply` *term*`,` *bindings_list* `,` `...` `,` *term*`,` *bindings_list* `in` *ident* `as` *disj_conj_intro_pattern* but using `eapply`.

6. `simple apply` *term* `in` *ident*

   This behaves like `apply` *term* `in` *ident* but it reasons modulo conversion only on subterms that contain no variables to instantiate. For instance, if `id := fun x:nat => x` and `H : forall y, id y = y -> True` and `H0 : O = O` then `simple apply H in H0` does not succeed because it would require the conversion of `f ?y` and `O` where `?y` is a variable to instantiate. Tactic `simple apply` *term* `in` *ident* does not either traverse tuples as `apply` *term* `in` *ident* does.

7. `simple apply` *term*`,` *bindings_list* `,` `...` `,` *term*`,` *bindings_list* `in` *ident* `as` *disj_conj_intro_pattern*
   `simple eapply` *term*`,` *bindings_list* `,` `...` `,` *term*`,` *bindings_list* `in` *ident* `as` *disj_conj_intro_pattern*

   This are the general forms of `simple apply` *term* `in` *ident* and `simple eapply` *term* `in` *ident*.

8.3.10 generalize *term*

This tactic applies to any goal. It generalizes the conclusion w.r.t. one subterm of it. For example:

```
Coq < Show.
1 subgoal

  x : nat
  y : nat
  ============================
   0 <= x + y + y

Coq < generalize (x + y + y).
1 subgoal

  x : nat
  y : nat
  ============================
   forall n : nat, 0 <= n
```

If the goal is G and t is a subterm of type T in the goal, then `generalize` *t* replaces the goal by `forall (x:`T`),` G' where G' is obtained from G by replacing all occurrences of t by `x`. The name of the variable (here `n`) is chosen based on T.

Variants:

1. `generalize` *term*$_1$ `, ... ,` *term*$_n$

   Is equivalent to `generalize` *term*$_n$`; ... ;` `generalize` *term*$_1$. Note that the sequence of *term*$_i$'s are processed from n to 1.

2. `generalize` *term* `at` *num*$_1$ `...` *num*$_i$

   Is equivalent to `generalize` *term* but generalizing only over the specified occurrences of *term* (counting from left to right on the expression printed using option `Set Printing All`).

3. `generalize` *term* `as` *ident*

   Is equivalent to `generalize` *term* but use *ident* to name the generalized hypothesis.

4. `generalize` *term*$_1$ `at` *num*$_{11}$ `...` *num*$_{1i_1}$ `as` *ident*$_1$ `, ...` `,` *term*$_n$ `at` *num*$_{n1}$ `...` *num*$_{ni_n}$ `as` *ident*$_2$

   This is the most general form of `generalize` that combines the previous behaviors.

5. `generalize dependent` *term*

   This generalizes *term* but also *all* hypotheses which depend on *term*. It clears the generalized hypotheses.

6. `revert` *ident*$_1$ `...` *ident*$_n$

   This is equivalent to a `generalize` followed by a `clear` on the given hypotheses. This tactic can be seen as reciprocal to `intros`.

8.3.11 `change` *term*

This tactic applies to any goal. It implements the rule "Conv" given in Section 4.3. `change U` replaces the current goal `T` with `U` providing that `U` is well-formed and that `T` and `U` are convertible.

Error messages:

1. `Not convertible`

Variants:

1. `change` *term*$_1$ `with` *term*$_2$

   This replaces the occurrences of *term*$_1$ by *term*$_2$ in the current goal. The terms *term*$_1$ and *term*$_2$ must be convertible.

2. `change` *term*$_1$ `at` *num*$_1$ `...` *num*$_i$ `with` *term*$_2$

   This replaces the occurrences numbered *num*$_1$ ... *num*$_i$ of *term*$_1$ by *term*$_2$ in the current goal. The terms *term*$_1$ and *term*$_2$ must be convertible.

   Error message: `Too few occurrences`

3. `change` *term* `in` *ident*

4. `change` *term*$_1$ `with` *term*$_2$ `in` *ident*

5. `change` *term*$_1$ `at` *num*$_1$ `...` *num*$_i$ `with` *term*$_2$ `in` *ident*

   This applies the `change` tactic not to the goal but to the hypothesis *ident*.

See also: 8.5

8.3.12 `fix` *ident num*

This tactic is a primitive tactic to start a proof by induction. In general, it is easier to rely on higher-level induction tactics such as the ones described in Section 8.7.

In the syntax of the tactic, the identifier *ident* is the name given to the induction hypothesis. The natural number *num* tells on which premise of the current goal the induction acts, starting from 1 and counting both dependent and non dependent products. Especially, the current lemma must be composed of at least *num* products.

Like in a `fix` expression, the induction hypotheses have to be used on structurally smaller arguments. The verification that inductive proof arguments are correct is done only at the time of registering the lemma in the environment. To know if the use of induction hypotheses is correct at some time of the interactive development of a proof, use the command `Guarded` (see Section 7.3.2).

Variants:

1. `fix` *ident*$_1$ *num* `with` `(` *ident*$_2$ *binder*$_2$ ... *binder*$_2$ `[{ struct` *ident*$'_2$ `}]` `:` *type*$_2$ `)` ... `(` *ident*$_1$ *binder*$_n$ ... *binder*$_n$ `[{ struct` *ident*$'_n$ `}]` `:` *type*$_n$ `)`

   This starts a proof by mutual induction. The statements to be simultaneously proved are respectively `forall` *binder*$_2$ ... *binder*$_2$`,` *type*$_2$, ..., `forall` *binder*$_n$ ... *binder*$_n$`,` *type*$_n$. The identifiers *ident*$_1$ ... *ident*$_n$ are the names of the induction hypotheses. The identifiers *ident*$'_2$ ... *ident*$'_n$ are the respective names of the premises on which the induction is performed in the statements to be simultaneously proved (if not given, the system tries to guess itself what they are).

8.3.13 `cofix` *ident*

This tactic starts a proof by coinduction. The identifier *ident* is the name given to the coinduction hypothesis. Like in a `cofix` expression, the use of induction hypotheses have to guarded by a constructor. The verification that the use of coinductive hypotheses is correct is done only at the time of registering the lemma in the environment. To know if the use of coinduction hypotheses is correct at some time of the interactive development of a proof, use the command `Guarded` (see Section 7.3.2).

Variants:

1. `cofix` $ident_1$ `with` `(` $ident_2$ $binder_2$ ... $binder_2$ `:` $type_2$ `)` ... `(` $ident_1$ $binder_1$ ... $binder_1$ `:` $type_n$ `)`

   This starts a proof by mutual coinduction. The statements to be simultaneously proved are respectively `forall` $binder_2$ ... $binder_2$`,` $type_2$, ..., `forall` $binder_n$ ... $binder_n$`,` $type_n$. The identifiers $ident_1$ ... $ident_n$ are the names of the coinduction hypotheses.

8.3.14 `evar` (*ident* : *term*)

The `evar` tactic creates a new local definition named *ident* with type *term* in the context. The body of this binding is a fresh existential variable.

8.3.15 `instantiate` (*num* := *term*)

The `instantiate` tactic allows to solve an existential variable with the term *term*. The *num* argument is the position of the existential variable from right to left in the conclusion. This cannot be the number of the existential variable since this number is different in every session.

Variants:

1. `instantiate` (*num* := *term*) `in` *ident*
2. `instantiate` (*num* := *term*) `in` (`Value of` *ident*)
3. `instantiate` (*num* := *term*) `in` (`Type of` *ident*)

   These allow to refer respectively to existential variables occurring in a hypothesis or in the body or the type of a local definition.

4. `instantiate`

   Without argument, the `instantiate` tactic tries to solve as many existential variables as possible, using information gathered from other tactics in the same tactical. This is automatically done after each complete tactic (i.e. after a dot in proof mode), but not, for example, between each tactic when they are sequenced by semicolons.

8.3.16 `admit`

The `admit` tactic "solves" the current subgoal by an axiom. This typically allows to temporarily skip a subgoal so as to progress further in the rest of the proof. To know if some proof still relies on unproved subgoals, one can use the command `Print Assumptions` (see Section 6.2.4). Admitted subgoals have names of the form *ident*`_admitted` possibly followed by a number.

8.3.17 Bindings list

Tactics that take a term as argument may also support a bindings list, so as to instantiate some parameters of the term by name or position. The general form of a term equipped with a bindings list is *term* `with` *bindings_list* where *bindings_list* may be of two different forms:

- In a bindings list of the form `(`*ref*$_1$ `:=` *term*$_1$`)` `...` `(`*ref*$_n$ `:=` *term*$_n$`)`, *ref* is either an *ident* or a *num*. The references are determined according to the type of *term*. If *ref*$_i$ is an identifier, this identifier has to be bound in the type of *term* and the binding provides the tactic with an instance for the parameter of this name. If *ref*$_i$ is some number n, this number denotes the n-th non dependent premise of the *term*, as determined by the type of *term*.

  Error message: `No such binder`

- A bindings list can also be a simple list of terms *term*$_1$ `...` *term*$_n$. In that case the references to which these terms correspond are determined by the tactic. In case of `induction`, `destruct`, `elim` and `case` (see Section 11) the terms have to provide instances for all the dependent products in the type of *term* while in the case of `apply`, or of `constructor` and its variants, only instances for the dependent products which are not bound in the conclusion of the type are required.

  Error message: `Not the right number of missing arguments`

8.3.18 Occurrences sets and occurrences clauses

An occurrences clause is a modifier to some tactics that obeys the following syntax:

occurrence_clause	::=	`in` *goal_occurrences*
goal_occurrences	::=	*[ident*$_1$ *[at_occurrences] ,*
		`...` `,`
		ident$_m$ *[at_occurrences]]*
		[`|-` *[*`*` *[at_occurrences]]]*
	`|`	`*` `|-` *[*`*` *[at_occurrences]]*
	`|`	`*`
at_occurrences	::=	`at` *occurrences*
occurrences	::=	*[-] num*$_1$ ... *num*$_n$

The role of an occurrence clause is to select a set of occurrences of a *term* in a goal. In the first case, the *ident*$_i$ *[*`at` *num*$_1^i$ ... *num*$_{n_i}^i$*]* parts indicate that occurrences have to be selected in the hypotheses named *ident*$_i$. If no numbers are given for hypothesis *ident*$_i$, then all occurrences of *term* in the hypothesis are selected. If numbers are given, they refer to occurrences of *term* when the term is printed using option `Set Printing All` (see Section 2.9), counting from left to right. In particular, occurrences of *term* in implicit arguments (see Section 2.7) or coercions (see Section 2.8) are counted.

If a minus sign is given between `at` and the list of occurrences, it negates the condition so that the clause denotes all the occurrences except the ones explicitly mentioned after the minus sign.

As an exception to the left-to-right order, the occurrences in the `return` subexpression of a `match` are considered *before* the occurrences in the matched term.

In the second case, the `*` on the left of `|-` means that all occurrences of *term* are selected in every hypothesis.

In the first and second case, if `*` is mentioned on the right of `|-`, the occurrences of the conclusion of the goal have to be selected. If some numbers are given, then only the occurrences denoted by these numbers are selected. In no numbers are given, all occurrences of *term* in the goal are selected.

Finally, the last notation is an abbreviation for * |- *. Note also that |- is optional in the first case when no * is given.

Here are some tactics that understand occurrences clauses: set, remember, induction, destruct.

See also: Sections 8.3.7, 8.7, 2.9.

8.4 Negation and contradiction

8.4.1 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 the current goal from False, and generates as subgoals ~P and P. It is very useful in proofs by cases, where some cases are impossible. In most cases, P or ~P is one of the hypotheses of the local context.

8.4.2 contradiction

This tactic applies to any goal. The contradiction tactic attempts to find in the current context (after all intros) one hypothesis 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 messages:

1. No such assumption

Variants:

1. contradiction *ident*

   The proof of False is searched in the hypothesis named *ident*.

8.4.3 contradict *ident*

This tactic allows to manipulate negated hypothesis and goals. The name *ident* should correspond to a hypothesis. With contradict H, the current goal and context is transformed in the following way:

- H:¬A ⊢ B becomes ⊢ A
- H:¬A ⊢ ¬B becomes H: B ⊢ A
- H: A ⊢ B becomes ⊢ ¬A
- H: A ⊢ ¬B becomes H: B ⊢ ¬A

8.5 Conversion tactics

This set of tactics implements different specialized usages of the tactic change.

All conversion tactics (including change) can be parameterized by the parts of the goal where the conversion can occur. This is done using *goal clauses* which consists in a list of hypotheses and,

optionally, of a reference to the conclusion of the goal. For defined hypothesis it is possible to specify if the conversion should occur on the type part, the body part or both (default).

Goal clauses are written after a conversion tactic (tactics `set` 8.3.7, `rewrite` 8.8.1, `replace` 8.8.3 and `autorewrite` 8.12.12 also use goal clauses) and are introduced by the keyword `in`. If no goal clause is provided, the default is to perform the conversion only in the conclusion.

The syntax and description of the various goal clauses is the following:

`in` *ident*$_1$ `...` *ident*$_n$ `|-` only in hypotheses *ident*$_1$ ... *ident*$_n$

`in` *ident*$_1$ `...` *ident*$_n$ `|-` `*` in hypotheses *ident*$_1$ ... *ident*$_n$ and in the conclusion

`in * |-` in every hypothesis

`in *` (equivalent to `in * |- *`) everywhere

`in (type of` *ident*$_1$`) (value of` *ident*$_2$`) ... |-` in type part of *ident*$_1$, in the value part of *ident*$_2$, etc.

For backward compatibility, the notation `in` *ident*$_1$... *ident*$_n$ performs the conversion in hypotheses *ident*$_1$... *ident*$_n$.

8.5.1 `cbv` *flag*$_1$ `...` *flag*$_n$, `lazy` *flag*$_1$ `...` *flag*$_n$ and `compute`

These parameterized reduction tactics apply to any goal and perform the normalization of the goal according to the specified flags. In correspondence with the kinds of reduction considered in Coq namely β (reduction of functional application), δ (unfolding of transparent constants, see 6.9.2), ι (reduction of pattern-matching over a constructed term, and unfolding of `fix` and `cofix` expressions) and ζ (contraction of local definitions), the flag are either `beta`, `delta`, `iota` or `zeta`. The `delta` flag itself can be refined into `delta [`*qualid*$_1$ ... *qualid*$_k$`]` or `delta -[`*qualid*$_1$ ... *qualid*$_k$`]`, restricting in the first case the constants to unfold to the constants listed, and restricting in the second case the constant to unfold to all but the ones explicitly mentioned. Notice that the `delta` flag does not apply to variables bound by a let-in construction inside the term itself (use here the `zeta` flag). In any cases, opaque constants are not unfolded (see Section 6.9.1).

The goal may be normalized with two strategies: *lazy* (`lazy` tactic), or *call-by-value* (`cbv` tactic). The lazy strategy is a call-by-need strategy, with sharing of reductions: the arguments of a function call are partially evaluated only when necessary, and if an argument is used several times then it is computed only once. This reduction is efficient for reducing expressions with dead code. For instance, the proofs of a proposition `exists` x`.` $P(x)$ reduce to a pair of a witness t, and a proof that t satisfies the predicate P. Most of the time, t may be computed without computing the proof of $P(t)$, thanks to the lazy strategy.

The call-by-value strategy is the one used in ML languages: the arguments of a function call are evaluated first, using a weak reduction (no reduction under the λ-abstractions). Despite the lazy strategy always performs fewer reductions than the call-by-value strategy, the latter is generally more efficient for evaluating purely computational expressions (i.e. with few dead code).

Variants:

1. `compute`
   `cbv`

   These are synonyms for `cbv beta delta iota zeta`.

2. `lazy`

   This is a synonym for `lazy beta delta iota zeta`.

3. `compute [`*qualid*$_1$ `...` *qualid*$_k$`]`
   `cbv [`*qualid*$_1$ `...` *qualid*$_k$`]`

   These are synonyms of `cbv beta delta [`*qualid*$_1$ `...` *qualid*$_k$`] iota zeta`.

4. `compute -[`*qualid*$_1$ `...` *qualid*$_k$`]`
   `cbv -[`*qualid*$_1$ `...` *qualid*$_k$`]`

   These are synonyms of `cbv beta delta -[`*qualid*$_1$ `...` *qualid*$_k$`] iota zeta`.

5. `lazy [`*qualid*$_1$ `...` *qualid*$_k$`]`
   `lazy -[`*qualid*$_1$ `...` *qualid*$_k$`]`

   These are respectively synonyms of `cbv beta delta [`*qualid*$_1$ `...` *qualid*$_k$`] iota zeta` and `cbv beta delta -[`*qualid*$_1$ `...` *qualid*$_k$`] iota zeta`.

6. `vm_compute`

   This tactic evaluates the goal using the optimized call-by-value evaluation bytecode-based virtual machine. This algorithm is dramatically more efficient than the algorithm used for the `cbv` tactic, but it cannot be fine-tuned. It is specially interesting for full evaluation of algebraic objects. This includes the case of reflexion-based tactics.

8.5.2 `red`

This tactic applies to a goal which has the form `forall (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 $\beta\iota\zeta$-reduction rules.

Error messages:

1. `Not reducible`

8.5.3 `hnf`

This tactic applies to any goal. It replaces the current goal with its head normal form according to the $\beta\delta\iota\zeta$-reduction rules, i.e. it reduces the head of the goal until it becomes a product or an irreducible term.

Example: The term `forall n:nat, (plus (S n) (S n))` is not reduced by `hnf`.

Remark: The δ rule only applies to transparent constants (see Section 6.9.1 on transparency and opacity).

8.5.4 `simpl`

This tactic applies to any goal. The tactic `simpl` first applies $\beta\iota$-reduction rule. 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` changes nothing. Notice that only transparent constants whose name can be reused as such in the recursive calls are possibly unfolded. For instance a constant defined by `plus' := plus` is possibly

unfolded and reused in the recursive calls, but a constant such as `succ := plus (S O)` is never unfolded.

Variants:

1. `simpl` *term*

   This applies `simpl` only to the occurrences of *term* in the current goal.

2. `simpl` *term* `at` num_1 `...` num_i

   This applies `simpl` only to the $num_1, \ldots, num_i$ occurrences of *term* in the current goal.

   Error message: `Too few occurrences`

3. `simpl` *ident*

   This applies `simpl` only to the applicative subterms whose head occurrence is *ident*.

4. `simpl` *ident* `at` num_1 `...` num_i

   This applies `simpl` only to the $num_1, \ldots, num_i$ applicative subterms whose head occurrence is *ident*.

8.5.5 `unfold` *qualid*

This tactic applies to any goal. The argument *qualid* must denote a defined transparent constant or local definition (see Sections 1.3.2 and 6.9.2). The tactic `unfold` applies the δ rule to each occurrence of the constant to which *qualid* refers in the current goal and then replaces it with its $\beta\iota$-normal form.

Error messages:

1. *qualid* `does not denote an evaluable constant`

Variants:

1. `unfold` $qualid_1$`,` `...`, $qualid_n$

   Replaces *simultaneously* $qualid_1, \ldots, qualid_n$ with their definitions and replaces the current goal with its $\beta\iota$ normal form.

2. `unfold` $qualid_1$ `at` num_1^1`,` `...`, num_i^1`,` `...`, $qualid_n$ `at` num_1^n `...` num_j^n

   The lists $num_1^1, \ldots, num_i^1$ and $num_1^n, \ldots, num_j^n$ specify the occurrences of $qualid_1, \ldots, qualid_n$ to be unfolded. Occurrences are located from left to right.

   Error message: `bad occurrence number of` $qualid_i$

   Error message: $qualid_i$ `does not occur`

3. `unfold` *string*

   If *string* denotes the discriminating symbol of a notation (e.g. `"+"`) or an expression defining a notation (e.g. `"_ + _"`), and this notation refers to an unfoldable constant, then the tactic unfolds it.

4. `unfold` *string*`%`*key*

   This is variant of `unfold` *string* where *string* gets its interpretation from the scope bound to the delimiting key *key* instead of its default interpretation (see Section 12.2.2).

5. `unfold` *qualid_or_string*$_1$ `at` *num*$_1^1$`, ..., `*num*$_i^1$`, ...,` *qualid_or_string*$_n$ `at` *num*$_1^n$ `...` *num*$_j^n$

   This is the most general form, where *qualid_or_string* is either a *qualid* or a *string* referring to a notation.

8.5.6 `fold` *term*

This tactic applies to any goal. The term *term* is reduced using the `red` tactic. Every occurrence of the resulting term in the goal is then replaced by *term*.

Variants:

1. `fold` *term*$_1$ ... *term*$_n$

   Equivalent to `fold` *term*$_1$`; ...; fold` *term*$_n$.

8.5.7 `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 (the inverse of β-reduction) 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 the abstracted goal to *term*

For instance, if the current goal T is expressible has $\phi(t)$ where the notation captures all the instances of t in $\phi(t)$, then `pattern` t transforms it into `(fun x:`A `=>` ϕ`(x))` t. This command can be used, for instance, when the tactic `apply` fails on matching.

Variants:

1. `pattern` *term* `at` *num*$_1$ `...` *num*$_n$

   Only the occurrences *num*$_1$ ... *num*$_n$ of *term* are considered for β-expansion. Occurrences are located from left to right.

2. `pattern` *term* `at -` *num*$_1$ `...` *num*$_n$

   All occurrences except the occurrences of indexes *num*$_1$ ... *num*$_n$ of *term* are considered for β-expansion. Occurrences are located from left to right.

3. `pattern` *term*$_1$`, ...,` *term*$_m$

   Starting from a goal $\phi(t_1 \ldots\ t_m)$, the tactic `pattern` t_1`, ...,` t_m generates the equivalent goal `(fun (x`$_1$`:`A_1`) ... (x`$_m$`:`A_m`) =>` ϕ`(x`$_1$`... x`$_m$`))` t_1 `...` t_m.
   If t_i occurs in one of the generated types A_j these occurrences will also be considered and possibly abstracted.

4. `pattern` *term*$_1$ `at` *num*$^1_1$ `...` *num*$^1_{n_1}$`,` `...,` *term*$_m$ `at` *num*$^m_1$ `...` *num*$^m_{n_m}$
   This behaves as above but processing only the occurrences *num*$^1_1, \ldots,$ *num*^1_i of *term*$_1, \ldots,$ *num*$^m_1$, $\ldots,$ *num*^m_j of *term*$_m$ starting from *term*$_m$.

5. `pattern` *term*$_1$ *[*`at` *[−] num*$^1_1 \ldots$ *num*$^1_{n_1}$*]* `,` $\ldots,$ *term*$_m$ *[*`at` *[−] num*$^m_1 \ldots$ *num*$^m_{n_m}$*]*
   This is the most general syntax that combines the different variants.

8.5.8 Conversion tactics applied to hypotheses

conv_tactic `in` *ident*$_1 \ldots$ *ident*$_n$

Applies the conversion tactic *conv_tactic* to the hypotheses *ident*$_1$, $\ldots$, *ident*$_n$. The tactic *conv_tactic* is any of the conversion tactics listed in this section.

If *ident*$_i$ is a local definition, then *ident*$_i$ can be replaced by (Type of *ident*$_i$) to address not the body but the type of the local definition. Example: `unfold not in (Type of H1) (Type of H3)`.

Error messages:

1. `No such hypothesis :` *ident*.

8.6 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.

8.6.1 `constructor` *num*

This tactic 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 messages:

1. `Not an inductive product`
2. `Not enough constructors`

Variants:

1. `constructor`
   This tries `constructor 1` then `constructor 2`, ... , then `constructor` *n* where *n* if the number of constructors of the head of the goal.

2. `constructor` *num* `with` *bindings_list*
   Let `ci` be the `i`-th constructor of `I`, then `constructor i with` *bindings_list* is equivalent to `intros; apply ci with` *bindings_list*.

   Warning: the terms in the *bindings_list* are checked in the context where `constructor` is executed and not in the context where `apply` is executed (the introductions are not taken into account).

3. `split`

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

4. `exists` *bindings_list*

   Applies if `I` has only one constructor, for instance in the case of existential quantification $\exists x \cdot P(x)$. Then, it is equivalent to `intros; constructor 1 with` *bindings_list*.

5. `left`
   `right`

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

6. `left` *bindings_list*
   `right` *bindings_list*
   `split` *bindings_list*

   As soon as the inductive type has the right number of constructors, these expressions are equivalent to the corresponding `constructor` i `with` *bindings_list*.

7. `econstructor`
   `eexists`
   `esplit`
   `eleft`
   `eright`

   These tactics and their variants behave like `constructor`, `exists`, `split`, `left`, `right` and their variants but they introduce existential variables instead of failing when the instantiation of a variable cannot be found (cf `eapply` and Section 10.2).

8.7 Induction and Case Analysis

The tactics presented in this section implement induction or case analysis on inductive or coinductive objects (see Section 4.5).

8.7.1 `induction` *term*

This tactic applies to any goal. The type of the argument *term* must be an inductive constant. Then, the tactic `induction` generates subgoals, one for each possible form of *term*, i.e. one for each constructor of the inductive type.

The tactic `induction` automatically replaces every occurrences of *term* in the conclusion and the hypotheses of the goal. It automatically adds induction hypotheses (using names of the form `IHn1`) to the local context. If some hypothesis must not be taken into account in the induction hypothesis, then it needs to be removed first (you can also use the tactics `elim` or `simple induction`, see below).

There are particular cases:

- If *term* is an identifier *ident* denoting a quantified variable of the conclusion of the goal, then `induction` *ident* behaves as `intros until` *ident*`;` `induction` *ident*.

- If *term* is a *num*, then `induction` *num* behaves as `intros until` *num* followed by `induction` applied to the last introduced hypothesis.

  Remark: For simple induction on a numeral, use syntax `induction (`*num*`)` (not very interesting anyway).

Example:

```
Coq < Lemma induction_test : forall n:nat, n = n -> n <= n.
1 subgoal

  ============================
   forall n : nat, n = n -> n <= n

Coq < intros n H.
1 subgoal

  n : nat
  H : n = n
  ============================
   n <= n

Coq < induction n.
2 subgoals

  H : 0 = 0
  ============================
   0 <= 0
subgoal 2 is:
 S n <= S n
```

Error messages:

1. `Not an inductive product`
2. `Unable to find an instance for the variables` *ident* `...` *ident*

   Use in this case the variant `elim ... with ...` below.

Variants:

1. `induction` *term* `as` *disj_conj_intro_pattern*

   This behaves as `induction` *term* but uses the names in *disj_conj_intro_pattern* to name the variables introduced in the context. The *disj_conj_intro_pattern* must typically be of the form `[` $p_{11} \ldots p_{1n_1}$ `|` `...` `|` $p_{m1} \ldots p_{mn_m}$ `]` with m being the number of constructors of the type of *term*. Each variable introduced by `induction` in the context of the i^{th} goal gets its name from the list $p_{i1} \ldots p_{in_i}$ in order. If there are not enough names, `induction` invents names for the remaining variables to introduce. More generally, the p_{ij} can be any disjunctive/conjunctive introduction pattern (see Section 8.7.3). For instance, for an inductive type with one constructor, the pattern notation `(`p_1`,...,`p_n`)` can be used instead of `[` $p_1 \ldots p_n$ `]`.

2. `induction` *term* `as` *naming_intro_pattern*

   This behaves as `induction` *term* but adds an equation between *term* and the value that *term* takes in each of the induction case. The name of the equation is built according to *naming_intro_pattern* which can be an identifier, a "?", etc, as indicated in Section 8.7.3.

3. `induction` *term* `as` *naming_intro_pattern* *disj_conj_intro_pattern*

   This combines the two previous forms.

4. `induction` *term* `with` *bindings_list*

   This behaves like `induction` *term* providing explicit instances for the premises of the type of *term* (see the syntax of bindings in Section 8.3.17).

5. `einduction` *term*

   This tactic behaves like `induction` *term* excepts that it does not fail if some dependent premise of the type of *term* is not inferable. Instead, the unresolved premises are posed as existential variables to be inferred later, in the same way as `eapply` does (see Section 10.2).

6. `induction` $term_1$ `using` $term_2$

   This behaves as `induction` $term_1$ but using $term_2$ as induction scheme. It does not expect the conclusion of the type of $term_1$ to be inductive.

7. `induction` $term_1$ `using` $term_2$ `with` *bindings_list*

   This behaves as `induction` $term_1$ `using` $term_2$ but also providing instances for the premises of the type of $term_2$.

8. `induction` $term_1$ `...` $term_n$ `using` *qualid*

   This syntax is used for the case *qualid* denotes an induction principle with complex predicates as the induction principles generated by `Function` or `Functional Scheme` may be.

9. `induction` *term* `in` *goal_occurrences*

   This syntax is used for selecting which occurrences of *term* the induction has to be carried on. The `in` *at_occurrences* clause is an occurrence clause whose syntax and behavior is described in Section 8.3.18.

   When an occurrence clause is given, an equation between *term* and the value it gets in each case of the induction is added to the context of the subgoals corresponding to the induction cases (even if no clause `as` *naming_intro_pattern* is given).

10. `induction` $term_1$ `with` $bindings\_list_1$ `as` *naming_intro_pattern* *disj_conj_intro_pattern* `using` $term_2$ `with` $bindings\_list_2$ `in` *goal_occurrences*
    `einduction` $term_1$ `with` $bindings\_list_1$ `as` *naming_intro_pattern* *disj_conj_intro_pattern* `using` $term_2$ `with` $bindings\_list_2$ `in` *goal_occurrences*

    These are the most general forms of `induction` and `einduction`. It combines the effects of the `with`, `as`, `using`, and `in` clauses.

11. `elim` *term*

    This is a more basic induction tactic. Again, the type of the argument *term* must be an inductive type. Then, according to the type of the goal, the tactic `elim` chooses the appropriate destructor

and applies it as the tactic `apply` would do. For instance, if the proof context contains `n:nat` and the current goal is `T` of type `Prop`, then `elim n` is equivalent to `apply nat_ind with (n:=n)`. The tactic `elim` does not modify the context of the goal, neither introduces the induction loading into the context of hypotheses.

More generally, `elim` *term* also works when the type of *term* is a statement with premises and whose conclusion is inductive. In that case the tactic performs induction on the conclusion of the type of *term* and leaves the non-dependent premises of the type as subgoals. In the case of dependent products, the tactic tries to find an instance for which the elimination lemma applies and fails otherwise.

12. `elim` *term* `with` *bindings_list*

    Allows to give explicit instances to the premises of the type of *term* (see Section 8.3.17).

13. `eelim` *term*

    In case the type of *term* has dependent premises, this turns them into existential variables to be resolved later on.

14. `elim` $term_1$ `using` $term_2$
    `elim` $term_1$ `using` $term_2$ `with` *bindings_list*

    Allows the user to give explicitly an elimination predicate $term_2$ which is not the standard one for the underlying inductive type of $term_1$. The *bindings_list* clause allows to instantiate premises of the type of $term_2$.

15. `elim` $term_1$ `with` $bindings\_list_1$ `using` $term_2$ `with` $bindings\_list_2$
    `eelim` $term_1$ `with` $bindings\_list_1$ `using` $term_2$ `with` $bindings\_list_2$

    These are the most general forms of `elim` and `eelim`. It combines the effects of the `using` clause and of the two uses of the `with` clause.

16. `elimtype` *form*

    The argument *form* must be inductively defined. `elimtype I` is equivalent to `cut I. intro H`*n*`; elim H`*n*`; clear H`*n*. Therefore 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` and which does not occur in the goal then `elim t` is equivalent to `elimtype I; 2: exact t.`

17. `simple induction` *ident*

    This tactic behaves as `intros until` *ident*`;` `elim` *ident* when *ident* is a quantified variable of the goal.

18. `simple induction` *num*

    This tactic behaves as `intros until` *num*`;` `elim` *ident* where *ident* is the name given by `intros until` *num* to the *num*-th non-dependent premise of the goal.

8.7.2 `destruct` *term*

The tactic `destruct` is used to perform case analysis without recursion. Its behavior is similar to `induction` except that no induction hypothesis is generated. It applies to any goal and the type of *term* must be inductively defined. There are particular cases:

- If *term* is an identifier *ident* denoting a quantified variable of the conclusion of the goal, then `destruct` *ident* behaves as `intros until` *ident*`;` `destruct` *ident*.
- If *term* is a *num*, then `destruct` *num* behaves as `intros until` *num* followed by `destruct` applied to the last introduced hypothesis.

  Remark: For destruction of a numeral, use syntax `destruct (`*num*`)` (not very interesting anyway).

Variants:

1. `destruct` *term* `as` *disj_conj_intro_pattern*

   This behaves as `destruct` *term* but uses the names in *intro_pattern* to name the variables introduced in the context. The *intro_pattern* must have the form `[` $p_{11} \ldots p_{1n_1}$ `|` $\ldots$ `|` $p_{m1} \ldots p_{mn_m}$ `]` with m being the number of constructors of the type of *term*. Each variable introduced by `destruct` in the context of the i^{th} goal gets its name from the list $p_{i1} \ldots p_{in_i}$ in order. If there are not enough names, `destruct` invents names for the remaining variables to introduce. More generally, the p_{ij} can be any disjunctive/conjunctive introduction pattern (see Section 8.7.3). This provides a concise notation for nested destruction.

2. `destruct` *term* `as` *disj_conj_intro_pattern* `_eqn`

   This behaves as `destruct` *term* but adds an equation between *term* and the value that *term* takes in each of the possible cases. The name of the equation is chosen by Coq. If *disj_conj_intro_pattern* is simply `[]`, it is automatically considered as a disjunctive pattern of the appropriate size.

3. `destruct` *term* `as` *disj_conj_intro_pattern* `_eqn:` *naming_intro_pattern*

   This behaves as `destruct` *term* `as` *disj_conj_intro_pattern* `_eqn` but use *naming_intro_pattern* to name the equation (see Section 8.7.3). Note that spaces can generally be removed around `_eqn`.

4. `destruct` *term* `with` *bindings_list*

   This behaves like `destruct` *term* providing explicit instances for the dependent premises of the type of *term* (see syntax of bindings in Section 8.3.17).

5. `edestruct` *term*

   This tactic behaves like `destruct` *term* excepts that it does not fail if the instance of a dependent premises of the type of *term* is not inferable. Instead, the unresolved instances are left as existential variables to be inferred later, in the same way as `eapply` does (see Section 10.2).

6. `destruct` $term_1$ `using` $term_2$
   `destruct` $term_1$ `using` $term_2$ `with` *bindings_list*

   These are synonyms of `induction` $term_1$ `using` $term_2$ and `induction` $term_1$ `using` $term_2$ `with` *bindings_list*.

7. `destruct` *term* `in` *goal_occurrences*

This syntax is used for selecting which occurrences of *term* the case analysis has to be done on. The `in` *goal_occurrences* clause is an occurrence clause whose syntax and behavior is described in Section 8.3.18.

When an occurrence clause is given, an equation between *term* and the value it gets in each case of the analysis is added to the context of the subgoals corresponding to the cases (even if no clause `as` *naming_intro_pattern* is given).

8. `destruct` *term*$_1$ `with` *bindings_list*$_1$ `as` *disj_conj_intro_pattern* `_eqn:` *naming_intro_pattern* `using` *term*$_2$ `with` *bindings_list*$_2$ `in` *goal_occurrences*
   `edestruct` *term*$_1$ `with` *bindings_list*$_1$ `as` *disj_conj_intro_pattern* `_eqn:` *naming_intro_pattern* `using` *term*$_2$ `with` *bindings_list*$_2$ `in` *goal_occurrences*

   These are the general forms of `destruct` and `edestruct`. They combine the effects of the `with`, `as`, `using`, and `in` clauses.

9. `case` *term*

   The tactic `case` is a more basic tactic to perform case analysis without recursion. It behaves as `elim` *term* but using a case-analysis elimination principle and not a recursive one.

10. `case_eq` *term*

    The tactic `case_eq` is a variant of the `case` tactic that allow to perform case analysis on a term without completely forgetting its original form. This is done by generating equalities between the original form of the term and the outcomes of the case analysis. The effect of this tactic is similar to the effect of `destruct` *term* `in |- *` with the exception that no new hypotheses are introduced in the context.

11. `case` *term* `with` *bindings_list*

    Analogous to `elim` *term* `with` *bindings_list* above.

12. `ecase` *term*
    `ecase` *term* `with` *bindings_list*

    In case the type of *term* has dependent premises, or dependent premises whose values are not inferable from the `with` *bindings_list* clause, `ecase` turns them into existential variables to be resolved later on.

13. `simple destruct` *ident*

    This tactic behaves as `intros until` *ident*`;` `case` *ident* when *ident* is a quantified variable of the goal.

14. `simple destruct` *num*

    This tactic behaves as `intros until` *num*`;` `case` *ident* where *ident* is the name given by `intros until` *num* to the *num*-th non-dependent premise of the goal.

8.7.3 `intros` *intro_pattern* `...` *intro_pattern*

This extension of the tactic `intros` combines introduction of variables or hypotheses and case analysis. An *introduction pattern* is either:

- A *naming introduction pattern*, i.e. either one of:

  - the pattern ?
  - the pattern ?*ident*
  - an identifier

- A *disjunctive/conjunctive introduction pattern*, i.e. either one of:
  - a disjunction of lists of patterns: [p_{11} ... p_{1m_1} | ... | p_{11} ... p_{nm_n}]
  - a conjunction of patterns: (p_1 , ... , p_n)
  - a list of patterns (p_1 & ... & p_n) for sequence of right-associative binary constructs
- the wildcard: _
- the rewriting orientations: -> or <-

Assuming a goal of type Q -> P (non dependent product), or of type forall x:T, P (dependent product), the behavior of intros p is defined inductively over the structure of the introduction pattern p:

- introduction on ? performs the introduction, and lets Coq choose a fresh name for the variable;
- introduction on ?*ident* performs the introduction, and lets Coq choose a fresh name for the variable based on *ident*;
- introduction on *ident* behaves as described in Section 8.3.5;
- introduction over a disjunction of list of patterns [p_{11} ... p_{1m_1} | ... | p_{11} ... p_{nm_n}] expects the product to be over an inductive type whose number of constructors is n (or more generally over a type of conclusion an inductive type built from n constructors, e.g. C -> A\/B if $n = 2$): it destructs the introduced hypothesis as destruct (see Section 8.7.2) would and applies on each generated subgoal the corresponding tactic; intros p_{i1} ... p_{im_i}; if the disjunctive pattern is part of a sequence of patterns and is not the last pattern of the sequence, then Coq completes the pattern so as all the argument of the constructors of the inductive type are introduced (for instance, the list of patterns [|] H applied on goal forall x:nat, x=0 -> 0=x behaves the same as the list of patterns [| ?] H);
- introduction over a conjunction of patterns (p_1, ..., p_n) expects the goal to be a product over an inductive type I with a single constructor that itself has at least n arguments: it performs a case analysis over the hypothesis, as destruct would, and applies the patterns p_1 ... p_n to the arguments of the constructor of I (observe that (p_1, ..., p_n) is an alternative notation for [p_1 ... p_n]);
- introduction via (p_1 & ...& p_n) is a shortcut for introduction via (p_1,(...,(...,p_n)...)); it expects the hypothesis to be a sequence of right-associative binary inductive constructors such as conj or ex_intro; for instance, an hypothesis with type A/\exists x, B/\C/\D can be introduced via pattern (a & x & b & c & d);
- introduction on the wildcard depends on whether the product is dependent or not: in the non dependent case, it erases the corresponding hypothesis (i.e. it behaves as an intro followed by a clear, cf Section 8.3.2) while in the dependent case, it succeeds and erases the variable only if the wildcard is part of a more complex list of introduction patterns that also erases the hypotheses depending on this variable;

- introduction over `->` (respectively `<-`) expects the hypothesis to be an equality and the right-hand-side (respectively the left-hand-side) is replaced by the left-hand-side (respectively the right-hand-side) in both the conclusion and the context of the goal; if moreover the term to substitute is a variable, the hypothesis is removed.

Remark: `intros` p_1 `...` p_n is not equivalent to `intros` p_1`;...;` `intros` p_n for the following reasons:

- A wildcard pattern never succeeds when applied isolated on a dependent product, while it succeeds as part of a list of introduction patterns if the hypotheses that depends on it are erased too.
- A disjunctive or conjunctive pattern followed by an introduction pattern forces the introduction in the context of all arguments of the constructors before applying the next pattern while a terminal disjunctive or conjunctive pattern does not. Here is an example

```
Coq < Goal forall n:nat, n = 0 -> n = 0.
1 subgoal

  ============================
   forall n : nat, n = 0 -> n = 0

Coq < intros [ | ] H.
2 subgoals

  H : 0 = 0
  ============================
   0 = 0
subgoal 2 is:
 S n = 0

Coq < Show 2.
subgoal 2 is:

  n : nat
  H : S n = 0
  ============================
   S n = 0

Coq < Undo.
1 subgoal

  ============================
   forall n : nat, n = 0 -> n = 0

Coq < intros [ | ]; intros H.
2 subgoals

  H : 0 = 0
  ============================
   0 = 0
subgoal 2 is:
 S H = 0 -> S H = 0

Coq < Show 2.
subgoal 2 is:
```

```
       H : nat
       ============================
        S H = 0 -> S H = 0

Coq < Lemma intros_test : forall A B C:Prop, A \/ B /\ C -> (A -> C) -> C.
1 subgoal

  ============================
   forall A B C : Prop, A \/ B /\ C -> (A -> C) -> C

Coq < intros A B C [a| [_ c]] f.
2 subgoals

  A : Prop
  B : Prop
  C : Prop
  a : A
  f : A -> C
  ============================
   C
subgoal 2 is:
 C

Coq < apply (f a).
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  c : C
  f : A -> C
  ============================
   C

Coq < exact c.
Proof completed.

Coq < Qed.
intros A B C [a| (_, c)] f.
 apply (f a).

 exact c.

intros_test is defined
```

8.7.4 `double induction` *ident*$_1$ *ident*$_2$

This tactic applies to any goal. If the variables *ident*$_1$ and *ident*$_2$ of the goal have an inductive type, then this tactic performs double induction on these variables. For instance, if the current goal is `forall n m:nat, P n m` then, `double induction n m` yields the four cases with their respective inductive hypotheses.

In particular, for proving `(P (S n) (S m))`, the generated induction hypotheses are `(P (S n) m)` and `(m:nat)(P n m)` (of the latter, `(P n m)` and `(P n (S m))` are derivable).

Remark: When the induction hypothesis `(P (S n) m)` is not needed, `induction` *ident*$_1$`;` `destruct` *ident*$_2$ produces more concise subgoals.

Variant:

1. `double induction` *num*$_1$ *num*$_2$

   This applies double induction on the *num*$_1^{th}$ and *num*$_2^{th}$ *non dependent* premises of the goal. More generally, any combination of an *ident* and a *num* is valid.

8.7.5 `dependent induction` *ident*

The *experimental* tactic `dependent induction` performs induction-inversion on an instantiated inductive predicate. One needs to first require the `Coq.Program.Equality` module to use this tactic. The tactic is based on the BasicElim tactic by Conor McBride [97] and the work of Cristina Cornes around inversion [35]. From an instantiated inductive predicate and a goal it generates an equivalent goal where the hypothesis has been generalized over its indexes which are then constrained by equalities to be the right instances. This permits to state lemmas without resorting to manually adding these equalities and still get enough information in the proofs. A simple example is the following:

```
Coq < Lemma le_minus : forall n:nat, n < 1 -> n = 0.
1 subgoal

  ============================
   forall n : nat, n < 1 -> n = 0

Coq < intros n H ; induction H.
2 subgoals

  n : nat
  ============================
   n = 0
subgoal 2 is:
 n = 0
```

Here we didn't get any information on the indexes to help fulfill this proof. The problem is that when we use the `induction` tactic we lose information on the hypothesis instance, notably that the second argument is `1` here. Dependent induction solves this problem by adding the corresponding equality to the context.

```
Coq < Require Import Coq.Program.Equality.

Coq < Lemma le_minus : forall n:nat, n < 1 -> n = 0.
1 subgoal

  ============================
   forall n : nat, n < 1 -> n = 0

Coq < intros n H ; dependent induction H.
2 subgoals

  ============================
   0 = 0
subgoal 2 is:
 n = 0
```

The subgoal is cleaned up as the tactic tries to automatically simplify the subgoals with respect to the generated equalities. In this enriched context it becomes possible to solve this subgoal.

```
Coq < reflexivity.
1 subgoal

  n : nat
  H : S n <= 0
  IHle : 0 = 1 -> n = 0
  ============================
   n = 0
```

Now we are in a contradictory context and the proof can be solved.

```
Coq < inversion H.
Proof completed.
```

This technique works with any inductive predicate. In fact, the `dependent induction` tactic is just a wrapper around the `induction` tactic. One can make its own variant by just writing a new tactic based on the definition found in `Coq.Program.Equality`. Common useful variants are the following, defined in the same file:

Variants:

1. `dependent induction` *ident* `generalizing` $ident_1$ `...` $ident_n$

   Does dependent induction on the hypothesis *ident* but first generalizes the goal by the given variables so that they are universally quantified in the goal. This is generally what one wants to do with the variables that are inside some constructors in the induction hypothesis. The other ones need not be further generalized.

2. `dependent destruction` *ident*

   Does the generalization of the instance *ident* but uses `destruct` instead of `induction` on the generalized hypothesis. This gives results equivalent to `inversion` or `dependent inversion` if the hypothesis is dependent.

A larger example of dependent induction and an explanation of the underlying technique are developed in section 10.6.

8.7.6 `decompose [` $qualid_1$ `...` $qualid_n$ `]` *term*

This tactic allows to recursively decompose a complex proposition in order to obtain atomic ones. Example:

```
Coq < Lemma ex1 : forall A B C:Prop, A /\ B /\ C \/ B /\ C \/ C /\ A -> C.
1 subgoal

  ============================
   forall A B C : Prop, A /\ B /\ C \/ B /\ C \/ C /\ A -> C

Coq < intros A B C H; decompose [and or] H; assumption.
Proof completed.

Coq < Qed.
```

`decompose` does not work on right-hand sides of implications or products.

Variants:

1. `decompose sum` *term* This decomposes sum types (like `or`).
2. `decompose record` *term* This decomposes record types (inductive types with one constructor, like `and` and `exists` and those defined with the `Record` macro, see Section 2.1).

8.7.7 `functional induction` (*qualid* $term_1$ ... $term_n$) .

The *experimental* tactic `functional induction` performs case analysis and induction following the definition of a function. It makes use of a principle generated by `Function` (see Section 2.3) or `Functional Scheme` (see Section 8.15).

```
Coq < Functional Scheme minus_ind := Induction for minus Sort Prop.
minus_equation is defined
minus_ind is defined

Coq <
Coq < Lemma le_minus : forall n m:nat, (n - m <= n).
1 subgoal

  ============================
   forall n m : nat, n - m <= n

Coq < intros n m.
1 subgoal

  n : nat
  m : nat
  ============================
   n - m <= n

Coq < functional induction (minus n m); simpl; auto.
Proof completed.

Coq < Qed.
```

Remark: (*qualid* $term_1$ ... $term_n$) must be a correct full application of *qualid*. In particular, the rules for implicit arguments are the same as usual. For example use `@`*qualid* if you want to write implicit arguments explicitly.

Remark: Parenthesis over *qualid*... $term_n$ are mandatory.

Remark: `functional induction (f x1 x2 x3)` is actually a wrapper for `induction x1 x2 x3 (f x1 x2 x3) using` *qualid* followed by a cleaning phase, where *qualid* is the induction principle registered for f (by the `Function` (see Section 2.3) or `Functional Scheme` (see Section 8.15) command) corresponding to the sort of the goal. Therefore `functional induction` may fail if the induction scheme (*qualid*) is not defined. See also Section 2.3 for the function terms accepted by `Function`.

Remark: There is a difference between obtaining an induction scheme for a function by using `Function` (see Section 2.3) and by using `Functional Scheme` after a normal definition using `Fixpoint` or `Definition`. See 2.3 for details.

See also: 2.3,8.15,10.4, 8.10.3

Error messages:

1. `Cannot find induction information on` *qualid*

2. `Not the right number of induction arguments`

Variants:

1. `functional induction (`*qualid* *term*$_1$ `...` *term*$_n$`) using` *term*$_{m+1}$ `with` *term*$_{n+1}$ `...`*term*$_m$

   Similar to `Induction` and `elim` (see Section 8.7), allows to give explicitly the induction principle and the values of dependent premises of the elimination scheme, including *predicates* for mutual induction when *qualid* is part of a mutually recursive definition.

2. `functional induction (`*qualid* *term*$_1$ `...` *term*$_n$`) using` *term*$_{m+1}$ `with` *ref*$_1$ `:=` *term*$_{n+1}$ `...` *ref*$_m$ `:=` *term*$_n$

   Similar to `induction` and `elim` (see Section 8.7).

3. All previous variants can be extended by the usual `as` *intro_pattern* construction, similar for example to `induction` and `elim` (see Section 8.7).

8.8 Equality

These tactics use the equality `eq:forall A:Type, A->A->Prop` defined in file `Logic.v` (see Section 3.1.2). The notation for `eq` T t u is simply t=u dropping the implicit type of t and u.

8.8.1 `rewrite` *term*

This tactic applies to any goal. The type of *term* must have the form
`forall (x`$_1$`:A`$_1$`) ... (x`$_n$`:A`$_n$`) eq` *term*$_1$ *term*$_2$.
where `eq` is the Leibniz equality or a registered setoid equality.
Then `rewrite` *term* finds the first subterm matching *term*$_1$ in the goal, resulting in instances *term*$'_1$ and *term*$'_2$ and then replaces every occurrence of *term*$'_1$ by *term*$'_2$. Hence, some of the variables x$_i$ are solved by unification, and some of the types `A`$_1$, ..., `A`$_n$ become new subgoals.

Error messages:

1. `The term provided does not end with an equation`

2. `Tactic generated a subgoal identical to the original goal`
   This happens if *term*$_1$ does not occur in the goal.

Variants:

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

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

3. `rewrite` *term* `in` *clause*
   Analogous to `rewrite` *term* but rewriting is done following *clause* (similarly to 8.5). For instance:

   - `rewrite H in H1` will rewrite `H` in the hypothesis `H1` instead of the current goal.
   - `rewrite H in H1 at 1, H2 at - 2 |- *` means `rewrite H; rewrite H in H1 at 1; rewrite H in H2 at - 2`. In particular a failure will happen if any of these three simpler tactics fails.
   - `rewrite H in * |-` will do `rewrite H in` H_i for all hypothesis H_i `<> H`. A success will happen as soon as at least one of these simpler tactics succeeds.
   - `rewrite H in *` is a combination of `rewrite H` and `rewrite H in * |-` that succeeds if at least one of these two tactics succeeds.

   Orientation `->` or `<-` can be inserted before the term to rewrite.

4. `rewrite` *term* `at` *occurrences*

   Rewrite only the given occurrences of $term'_1$. Occurrences are specified from left to right as for `pattern` (§8.5.7). The rewrite is always performed using setoid rewriting, even for Leibniz's equality, so one has to `Import Setoid` to use this variant.

5. `rewrite` *term* `by` *tactic*

   Use *tactic* to completely solve the side-conditions arising from the rewrite.

6. `rewrite` $term_1$`, ...,` $term_n$
   Is equivalent to the n successive tactics `rewrite` $term_1$ up to `rewrite` $term_n$, each one working on the first subgoal generated by the previous one. Orientation `->` or `<-` can be inserted before each term to rewrite. One unique *clause* can be added at the end after the keyword `in`; it will then affect all rewrite operations.

7. In all forms of `rewrite` described above, a term to rewrite can be immediately prefixed by one of the following modifiers:

   - `?` : the tactic `rewrite ?`*term* performs the rewrite of *term* as many times as possible (perhaps zero time). This form never fails.
   - n`?` : works similarly, except that it will do at most n rewrites.
   - `!` : works as `?`, except that at least one rewrite should succeed, otherwise the tactic fails.
   - n`!` (or simply n) : precisely n rewrites of *term* will be done, leading to failure if these n rewrites are not possible.

8. `erewrite` *term*

   This tactic works as `rewrite` *term* but turning unresolved bindings into existential variables, if any, instead of failing. It has the same variants as `rewrite` has.

8.8.2 `cutrewrite ->` $term_1$ `=` $term_2$

This tactic acts like `replace` $term_1$ `with` $term_2$ (see below).

8.8.3 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. This equality is automatically solved if it occurs amongst the assumption, or if its symmetric form occurs. It is equivalent to `cut` *term*$_2$=*term*$_1$`; [intro H`*n*`; rewrite <- H`*n*`; clear H`*n*`| assumption || symmetry; try assumption]`.

Error messages:

1. `terms do not have convertible types`

Variants:

1. `replace` *term*$_1$ `with` *term*$_2$ `by` *tactic*
   This acts as `replace` *term*$_1$ `with` *term*$_2$ but applies *tactic* to solve the generated subgoal *term*$_2$=*term*$_1$.

2. `replace` *term*
   Replace *term* with *term*' using the first assumption whose type has the form *term*=*term*' or *term*'=*term*

3. `replace ->` *term*
   Replace *term* with *term*' using the first assumption whose type has the form *term*=*term*'

4. `replace <-` *term*
   Replace *term* with *term*' using the first assumption whose type has the form *term*'=*term*

5. `replace` *term*$_1$ `with` *term*$_2$ *`clause`*
   `replace` *term*$_1$ `with` *term*$_2$ *`clause`* `by` *tactic*
   `replace` *term* *`clause`*
   `replace ->` *term* *`clause`*
   `replace <-` *term* *`clause`*
   Act as before but the replacements take place in *clause* (see Section 8.5) and not only in the conclusion of the goal.
   The *clause* argument must not contain any `type of` nor `value of`.

8.8.4 reflexivity

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

Error messages:

1. `The conclusion is not a substitutive equation`
2. `Impossible to unify ... with ....`

8.8.5 symmetry

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

Variant: `symmetry in` *ident*
If the statement of the hypothesis *ident* has the form `t=u`, the tactic changes it to `u=t`.

8.8.6 `transitivity` *term*

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

8.8.7 `subst` *ident*

This tactic applies to a goal which has *ident* in its context and (at least) one hypothesis, say `H`, of type *ident*`=t` or `t=`*ident*. Then it replaces *ident* by `t` everywhere in the goal (in the hypotheses and in the conclusion) and clears *ident* and `H` from the context.

Remark: When several hypotheses have the form *ident*`=t` or `t=`*ident*, the first one is used.

Variants:

1. `subst` $ident_1$ `...` $ident_n$
   Is equivalent to `subst` $ident_1$`; ...; subst` $ident_n$.

2. `subst`
   Applies `subst` repeatedly to all identifiers from the context for which an equality exists.

8.8.8 `stepl` *term*

This tactic is for chaining rewriting steps. It assumes a goal of the form "R *term*$_1$ *term*$_2$" where R is a binary relation and relies on a database of lemmas of the form `forall` $x\ y\ z$, $R\ x\ y$ `->` $eq\ x\ z$ `->` $R\ z\ y$ where eq is typically a setoid equality. The application of `stepl` *term* then replaces the goal by "R *term term*$_2$" and adds a new goal stating "eq *term term*$_1$".

Lemmas are added to the database using the command

`Declare Left Step` *term*`.`

The tactic is especially useful for parametric setoids which are not accepted as regular setoids for `rewrite` and `setoid_replace` (see Chapter 24).

Variants:

1. `stepl` *term* `by` *tactic*
   This applies `stepl` *term* then applies *tactic* to the second goal.

2. `stepr` *term*
   `stepr` *term* `by` *tactic*
   This behaves as `stepl` but on the right-hand-side of the binary relation. Lemmas are expected to be of the form "`forall` $x\ y\ z$, $R\ x\ y$ `->` $eq\ y\ z$ `->` $R\ x\ z$" and are registered using the command

   `Declare Right Step` *term*`.`

8.8.9 `f_equal`

This tactic applies to a goal of the form $f\ a_1\ \ldots\ a_n = f'\ a'_1\ \ldots\ a'_n$. Using `f_equal` on such a goal leads to subgoals $f = f'$ and $a_1 = a'_1$ and so on up to $a_n = a'_n$. Amongst these subgoals, the simple ones (e.g. provable by reflexivity or congruence) are automatically solved by `f_equal`.

8.9 Equality and inductive sets

We describe in this section some special purpose tactics dealing with equality and inductive sets or types. These tactics use the equality `eq:forall (A:Type), A->A->Prop`, simply written with the infix symbol =.

8.9.1 `decide equality`

This tactic solves a goal of the form `forall x y:`R`, {`x`=`y`}+{~`x`=`y`}`, where R is an inductive type such that its constructors do not take proofs or functions as arguments, nor objects in dependent types.

Variants:

1. `decide equality` *term*$_1$ *term*$_2$.
   Solves a goal of the form {*term*$_1$=*term*$_2$}+{~*term*$_1$=*term*$_2$}.

8.9.2 `compare` *term*$_1$ *term*$_2$

This tactic compares two given objects *term*$_1$ and *term*$_2$ of an inductive datatype. If G is the current goal, it leaves the sub-goals *term*$_1$=*term*$_2$ -> G and ~*term*$_1$=*term*$_2$ -> G. The type of *term*$_1$ and *term*$_2$ must satisfy the same restrictions as in the tactic `decide equality`.

8.9.3 `discriminate` *term*

This tactic proves any goal from an assumption stating that two structurally different terms of an inductive set are equal. For example, from `(S (S O))=(S O)` we can derive by absurdity any proposition.

The argument *term* is assumed to be a proof of a statement of conclusion *term*$_1$ = *term*$_2$ with *term*$_1$ and *term*$_2$ being elements of an inductive set. To build the proof, the tactic traverses the normal forms[4] 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 at the same positions and whose head symbols are two different constructors. If such a couple of subterms exists, then the proof of the current goal is completed, otherwise the tactic fails.

Remark: The syntax `discriminate` *ident* can be used to refer to a hypothesis quantified in the goal. In this case, the quantified hypothesis whose name is *ident* is first introduced in the local context using `intros until` *ident*.

Error messages:

1. `No primitive equality found`
2. `Not a discriminable equality`

Variants:

1. `discriminate` *num*

   This does the same thing as `intros until` *num* followed by `discriminate` *ident* where *ident* is the identifier for the last introduced hypothesis.

[4] Reminder: opaque constants will not be expanded by δ reductions

2. `discriminate` *term* `with` *bindings_list*

   This does the same thing as `discriminate` *term* but using the given bindings to instantiate parameters or hypotheses of *term*.

3. `ediscriminate` *num*
   `ediscriminate` *term* [`with` *bindings_list*]

   This works the same as `discriminate` but if the type of *term*, or the type of the hypothesis referred to by *num*, has uninstantiated parameters, these parameters are left as existential variables.

4. `discriminate`

   This behaves like `discriminate` *ident* if *ident* is the name of an hypothesis to which `discriminate` is applicable; if the current goal is of the form $term_1$ <> $term_2$, this behaves as `intro` *ident*`;` `injection` *ident*.

   Error messages:

   (a) `No discriminable equalities`
   occurs when the goal does not verify the expected preconditions.

8.9.4 `injection` *term*

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 if $(c\ \vec{t_1})$ and $(c\ \vec{t_2})$ are two terms that are equal then $\vec{t_1}$ and $\vec{t_2}$ are equal too.

If *term* is a proof of a statement of conclusion $term_1$ = $term_2$, then `injection` applies injectivity 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 `(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 n_1 and n_2 are their respective normal forms, then:

- n_1 and n_2 should not be syntactically equal,
- there must not exist any pair 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.

Example: Consider the following goal:

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

Coq < Variable P : list -> Prop.

Coq < Show.
1 subgoal

  l : list
  n : nat
```

```
  H : P nil
  H0 : cons n l = cons 0 nil
  ============================
   P l

Coq < injection H0.
1 subgoal

  l : list
  n : nat
  H : P nil
  H0 : cons n l = cons 0 nil
  ============================
   l = nil -> n = 0 -> P l
```

Beware that `injection` yields always an equality in a sigma type whenever the injected object has a dependent type.

Remark: There is a special case for dependent pairs. If we have a decidable equality over the type of the first argument, then it is safe to do the projection on the second one, and so `injection` will work fine. To define such an equality, you have to use the `Scheme` command (see 8.14).

Remark: If some quantified hypothesis of the goal is named *ident*, then `injection` *ident* first introduces the hypothesis in the local context using `intros until` *ident*.

Error messages:

1. `Not a projectable equality but a discriminable one`
2. `Nothing to do, it is an equality between convertible terms`
3. `Not a primitive equality`

Variants:

1. `injection` *num*

   This does the same thing as `intros until` *num* followed by `injection` *ident* where *ident* is the identifier for the last introduced hypothesis.

2. `injection` *term* `with` *bindings_list*

   This does the same as `injection` *term* but using the given bindings to instantiate parameters or hypotheses of *term*.

3. `einjection` *num*
   `einjection` *term* [`with` *bindings_list*]

   This works the same as `injection` but if the type of *term*, or the type of the hypothesis referred to by *num*, has uninstantiated parameters, these parameters are left as existential variables.

4. `injection`

   If the current goal is of the form $term_1$ <> $term_2$, this behaves as `intro` *ident*`;` `injection` *ident*.

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

5. `injection` *term* [`with` *bindings_list*] `as` *intro_pattern* ... *intro_pattern*
   `injection` *num* `as` *intro_pattern* ... *intro_pattern*
   `injection as` *intro_pattern* ... *intro_pattern*
   `einjection` *term* [`with` *bindings_list*] `as` *intro_pattern* ... *intro_pattern*
   `einjection` *num* `as` *intro_pattern* ... *intro_pattern*
   `einjection as` *intro_pattern* ... *intro_pattern*

   These variants apply `intros` *intro_pattern* ... *intro_pattern* after the call to `injection` or `einjection`.

8.9.5 `simplify_eq` *term*

Let *term* be the proof of a statement of conclusion $term_1$=$term_2$. If $term_1$ and $term_2$ are structurally different (in the sense described for the tactic `discriminate`), then the tactic `simplify_eq` behaves as `discriminate` *term*, otherwise it behaves as `injection` *term*.

Remark: If some quantified hypothesis of the goal is named *ident*, then `simplify_eq` *ident* first introduces the hypothesis in the local context using `intros until` *ident*.

Variants:

1. `simplify_eq` *num*

   This does the same thing as `intros until` *num* then `simplify_eq` *ident* where *ident* is the identifier for the last introduced hypothesis.

2. `simplify_eq` *term* `with` *bindings_list*

   This does the same as `simplify_eq` *term* but using the given bindings to instantiate parameters or hypotheses of *term*.

3. `esimplify_eq` *num*
   `esimplify_eq` *term* [`with` *bindings_list*]

   This works the same as `simplify_eq` but if the type of *term*, or the type of the hypothesis referred to by *num*, has uninstantiated parameters, these parameters are left as existential variables.

4. `simplify_eq`

   If the current goal has form t_1<><t_2, it behaves as `intro` *ident*`;` `simplify_eq` *ident*.

8.9.6 `dependent rewrite ->` *ident*

This tactic applies to any goal. If *ident* has type `(existT B a b)=(existT B a' b')` in the local context (i.e. each term of the equality has a sigma type $\{a : A \,\&\, (B\ a)\}$) this tactic rewrites `a` into `a'` and `b` into `b'` in the current goal. This tactic works even if B is also a sigma type. This kind of equalities between dependent pairs may be derived by the injection and inversion tactics.

Variants:

1. `dependent rewrite <-` *ident*
   Analogous to `dependent rewrite ->` but uses the equality from right to left.

8.10 Inversion

8.10.1 `inversion` *ident*

Let the type of *ident* in the local context be $(I\ \vec{t})$, where I is a (co)inductive predicate. Then, `inversion` applied to *ident* derives for each possible constructor c_i of $(I\ \vec{t})$, **all** the necessary conditions that should hold for the instance $(I\ \vec{t})$ to be proved by c_i.

Remark: If *ident* does not denote a hypothesis in the local context but refers to a hypothesis quantified in the goal, then the latter is first introduced in the local context using `intros until` *ident*.

Variants:

1. `inversion` *num*

   This does the same thing as `intros until` *num* then `inversion` *ident* where *ident* is the identifier for the last introduced hypothesis.

2. `inversion_clear` *ident*

   This behaves as `inversion` and then erases *ident* from the context.

3. `inversion` *ident* `as` *intro_pattern*

   This behaves as `inversion` but using names in *intro_pattern* for naming hypotheses. The *intro_pattern* must have the form `[` $p_{11} \ldots p_{1n_1}$ `|` $\ldots$ `|` $p_{m1} \ldots p_{mn_m}$ `]` with m being the number of constructors of the type of *ident*. Be careful that the list must be of length m even if `inversion` discards some cases (which is precisely one of its roles): for the discarded cases, just use an empty list (i.e. $n_i = 0$).

   The arguments of the i^{th} constructor and the equalities that `inversion` introduces in the context of the goal corresponding to the i^{th} constructor, if it exists, get their names from the list $p_{i1} \ldots p_{in_i}$ in order. If there are not enough names, `induction` invents names for the remaining variables to introduce. In case an equation splits into several equations (because `inversion` applies `injection` on the equalities it generates), the corresponding name p_{ij} in the list must be replaced by a sublist of the form `[`p_{ij1} `...`p_{ijq}`]` (or, equivalently, `(`p_{ij1}`, ..., `p_{ijq}`)`) where q is the number of subequalities obtained from splitting the original equation. Here is an example.

```
Coq < Inductive contains0 : list nat -> Prop :=
Coq <   | in_hd : forall l, contains0 (0 :: l)
Coq <   | in_tl : forall l b, contains0 l -> contains0 (b :: l).
contains0 is defined
contains0_ind is defined

Coq < Goal forall l:list nat, contains0 (1 :: l) -> contains0 l.
1 subgoal

  ============================
   forall l : list nat, contains0 (1 :: l) -> contains0 l

Coq < intros l H; inversion H as [ | l' p Hl' [Heqp Heql'] ].
1 subgoal

  l : list nat
  H : contains0 (1 :: l)
```

```
l' : list nat
p : nat
Hl' : contains0 l
Heqp : p = 1
Heql' : l' = l
============================
 contains0 l
```

4. `inversion` *num* `as` *intro_pattern*

   This allows to name the hypotheses introduced by `inversion` *num* in the context.

5. `inversion_clear` *ident* `as` *intro_pattern*

   This allows to name the hypotheses introduced by `inversion_clear` in the context.

6. `inversion` *ident* `in` $ident_1 \ldots ident_n$

   Let $ident_1 \ldots ident_n$, be identifiers in the local context. This tactic behaves as generalizing $ident_1 \ldots ident_n$, and then performing `inversion`.

7. `inversion` *ident* `as` *intro_pattern* `in` $ident_1 \ldots ident_n$

   This allows to name the hypotheses introduced in the context by `inversion` *ident* `in` $ident_1 \ldots ident_n$.

8. `inversion_clear` *ident* `in` $ident_1 \ldots ident_n$

   Let $ident_1 \ldots ident_n$, be identifiers in the local context. This tactic behaves as generalizing $ident_1 \ldots ident_n$, and then performing `inversion_clear`.

9. `inversion_clear` *ident* `as` *intro_pattern* `in` $ident_1 \ldots ident_n$

   This allows to name the hypotheses introduced in the context by `inversion_clear` *ident* `in` $ident_1 \ldots ident_n$.

10. `dependent inversion` *ident*

    That must be used when *ident* appears in the current goal. It acts like `inversion` and then substitutes *ident* for the corresponding term in the goal.

11. `dependent inversion` *ident* `as` *intro_pattern*

    This allows to name the hypotheses introduced in the context by `dependent inversion` *ident*.

12. `dependent inversion_clear` *ident*

    Like `dependent inversion`, except that *ident* is cleared from the local context.

13. `dependent inversion_clear` *ident* `as` *intro_pattern*

    This allows to name the hypotheses introduced in the context by `dependent inversion_clear` *ident*.

14. `dependent inversion` *ident* `with` *term*

    This variant allows you to specify the generalization of the goal. It is useful when the system fails to generalize the goal automatically. If *ident* has type $(I\ \vec{t})$ and I has type $forall(\vec{x} : \vec{T}), s$, then *term* must be of type $I : forall(\vec{x} : \vec{T}), I\ \vec{x} \rightarrow s'$ where s' is the type of the goal.

15. `dependent inversion` *ident* `as` *intro_pattern* `with` *term*

    This allows to name the hypotheses introduced in the context by `dependent inversion` *ident* `with` *term*.

16. `dependent inversion_clear` *ident* `with` *term*

    Like `dependent inversion ... with` but clears *ident* from the local context.

17. `dependent inversion_clear` *ident* `as` *intro_pattern* `with` *term*

    This allows to name the hypotheses introduced in the context by `dependent inversion_clear` *ident* `with` *term*.

18. `simple inversion` *ident*

    It is a very primitive inversion tactic that derives all the necessary equalities but it does not simplify the constraints as `inversion` does.

19. `simple inversion` *ident* `as` *intro_pattern*

    This allows to name the hypotheses introduced in the context by `simple inversion`.

20. `inversion` *ident* `using` *ident′*

    Let *ident* have type $(I\ \vec{t})$ (I an inductive predicate) in the local context, and *ident′* be a (dependent) inversion lemma. Then, this tactic refines the current goal with the specified lemma.

21. `inversion` *ident* `using` *ident′* `in` $ident_1 \ldots\ ident_n$

    This tactic behaves as generalizing $ident_1 \ldots\ ident_n$, then doing `inversion` *ident* `using` *ident′*.

See also: 10.5 for detailed examples

8.10.2 `Derive Inversion` *ident* `with forall`$(\vec{x}{:}\vec{T}),I\ \vec{t}$ `Sort` *sort*

This command generates an inversion principle for the `inversion ... using` tactic. Let I be an inductive predicate and $\vec{x}$ the variables occurring in $\vec{t}$. This command generates and stocks the inversion lemma for the sort *sort* corresponding to the instance $forall(\vec{x} : \vec{T}), I\ \vec{t}$ with the name *ident* in the **global** environment. When applied it is equivalent to have inverted the instance with the tactic `inversion`.

Variants:

1. `Derive Inversion_clear` *ident* `with` $forall(\vec{x} : \vec{T}), I\ \vec{t}$ `Sort` *sort*
   When applied it is equivalent to having inverted the instance with the tactic `inversion` replaced by the tactic `inversion_clear`.

2. `Derive Dependent Inversion` *ident* `with` $forall(\vec{x} : \vec{T}), I\ \vec{t}$ `Sort` *sort*
   When applied it is equivalent to having inverted the instance with the tactic `dependent inversion`.

3. `Derive Dependent Inversion_clear` *ident* `with` $forall(\vec{x} : \vec{T}), I\ \vec{t}$ `Sort` *sort*
   When applied it is equivalent to having inverted the instance with the tactic `dependent inversion_clear`.

See also: 10.5 for examples

8.10.3 `functional inversion` *ident*

`functional inversion` is a *highly* experimental tactic which performs inversion on hypothesis *ident* of the form *qualid* $term_1 \ldots term_n$ = *term* or *term* = *qualid* $term_1 \ldots term_n$ where *qualid* must have been defined using `Function` (see Section 2.3).

Error messages:

1. `Hypothesis` *ident* `must contain at least one Function`
2. `Cannot find inversion information for hypothesis` *ident* This error may be raised when some inversion lemma failed to be generated by Function.

Variants:

1. `functional inversion` *num*

   This does the same thing as `intros until` *num* then `functional inversion` *ident* where *ident* is the identifier for the last introduced hypothesis.

2. `functional inversion` *ident* *qualid*
   `functional inversion` *num* *qualid*

   In case the hypothesis *ident* (or *num*) has a type of the form $qualid_1$ $term_1 \ldots term_n$ = $qualid_2$ $term_{n+1} \ldots term_{n+m}$ where $qualid_1$ and $qualid_2$ are valid candidates to functional inversion, this variant allows to choose which must be inverted.

8.10.4 `quote` *ident*

This kind of inversion has nothing to do with the tactic `inversion` above. This tactic does `change (`*ident* `t)`, where `t` is a term built in order to ensure the convertibility. In other words, it does inversion of the function *ident*. This function must be a fixpoint on a simple recursive datatype: see 10.8 for the full details.

Error messages:

1. `quote: not a simple fixpoint`
   Happens when `quote` is not able to perform inversion properly.

Variants:

1. `quote` *ident* `[` $ident_1 \ldots ident_n$ `]`
   All terms that are built only with $ident_1 \ldots ident_n$ will be considered by `quote` as constants rather than variables.

8.11 Classical tactics

In order to ease the proving process, when the `Classical` module is loaded. A few more tactics are available. Make sure to load the module using the `Require Import` command.

8.11.1 classical_left, classical_right

The tactics classical_left and classical_right are the analog of the left and right but using classical logic. They can only be used for disjunctions. Use classical_left to prove the left part of the disjunction with the assumption that the negation of right part holds. Use classical_right to prove the right part of the disjunction with the assumption that the negation of left part holds.

8.12 Automatizing

8.12.1 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.

By default, auto only uses the hypotheses of the current goal and the hints of the database named core.

Variants:

1. auto *num*

   Forces the search depth to be *num*. The maximal search depth is 5 by default.

2. auto with $ident_1$ ... $ident_n$

   Uses the hint databases $ident_1$ ... $ident_n$ in addition to the database core. See Section 8.13.1 for the list of pre-defined databases and the way to create or extend a database. This option can be combined with the previous one.

3. auto with *

   Uses all existing hint databases, minus the special database v62. See Section 8.13.1

4. auto using $lemma_1, \ldots, lemma_n$

   Uses $lemma_1, \ldots, lemma_n$ in addition to hints (can be combined with the with *ident* option).

5. trivial

   This tactic is a restriction of auto that is not recursive and tries only hints which cost 0. Typically it solves trivial equalities like $X = X$.

6. trivial with $ident_1$ ... $ident_n$

7. trivial with *

Remark: auto either solves completely the goal or else leaves it intact. auto and trivial never fail.

See also: Section 8.13.1

8.12.2 eauto

This tactic generalizes auto. In contrast with the latter, eauto uses unification of the goal against the hints rather than pattern-matching (in other words, it uses eapply instead of apply). As a consequence, eauto can solve such a goal:

```
Coq < Hint Resolve ex_intro.
Warning: the hint: eapply ex_intro will only be used by eauto

Coq < Goal forall P:nat -> Prop, P 0 ->  exists n, P n.
1 subgoal

  ============================
   forall P0 : nat -> Prop, P0 0 -> exists n : nat, P0 n

Coq < eauto.
Proof completed.
```

Note that ex_intro should be declared as an hint.

See also: Section 8.13.1

8.12.3 tauto

This tactic implements a decision procedure for intuitionistic propositional calculus based on the contraction-free sequent calculi LJT* of Roy Dyckhoff [54]. Note that tauto succeeds on any instance of an intuitionistic tautological proposition. tauto unfolds negations and logical equivalence but does not unfold any other definition.

The following goal can be proved by tauto whereas auto would fail:

```
Coq < Goal forall (x:nat) (P:nat -> Prop), x = 0 \/ P x -> x <> 0 -> P x.
1 subgoal

  ============================
   forall (x : nat) (P0 : nat -> Prop), x = 0 \/ P0 x -> x <> 0 -> P0 x

Coq <   intros.
1 subgoal

  x : nat
  P0 : nat -> Prop
  H : x = 0 \/ P0 x
  H0 : x <> 0
  ============================
   P0 x

Coq <   tauto.
Proof completed.
```

Moreover, if it has nothing else to do, tauto performs introductions. Therefore, the use of intros in the previous proof is unnecessary. tauto can for instance prove the following:

```
Coq < (* auto would fail *)
Coq < Goal forall (A:Prop) (P:nat -> Prop),
Coq <     A \/ (forall x:nat, ~ A -> P x) -> forall x:nat, ~ A -> P x.
```

```
1 subgoal

  ============================
   forall (A : Prop) (P0 : nat -> Prop),
   A \/ (forall x : nat, ~ A -> P0 x) -> forall x : nat, ~ A -> P0 x

Coq <
Coq <   tauto.
Proof completed.
```

Remark: In contrast, `tauto` cannot solve the following goal

```
Coq < Goal forall (A:Prop) (P:nat -> Prop),
Coq <     A \/ (forall x:nat, ~ A -> P x) -> forall x:nat, ~ ~ (A \/ P x).
```

because `(forall x:nat, ~ A -> P x)` cannot be treated as atomic and an instantiation of `x` is necessary.

8.12.4 `intuition` *tactic*

The tactic `intuition` takes advantage of the search-tree built 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 *tactic* to them [103]. If this tactic fails on some goals then `intuition` fails. In fact, `tauto` is simply `intuition fail`.

For instance, the tactic `intuition auto` applied to the goal

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

internally replaces it by the equivalent one:

```
(forall (x:nat), P x), B |- P O
```

and then uses `auto` which completes the proof.

Originally due to César Muñoz, these tactics (`tauto` and `intuition`) have been completely re-engineered by David Delahaye using mainly the tactic language (see Chapter 9). The code is now much shorter and a significant increase in performance has been noticed. The general behavior with respect to dependent types, unfolding and introductions has slightly changed to get clearer semantics. This may lead to some incompatibilities.

Variants:

1. `intuition`
   Is equivalent to `intuition auto with *`.

8.12.5 `rtauto`

The `rtauto` tactic solves propositional tautologies similarly to what `tauto` does. The main difference is that the proof term is built using a reflection scheme applied to a sequent calculus proof of the goal. The search procedure is also implemented using a different technique.

Users should be aware that this difference may result in faster proof-search but slower proof-checking, and `rtauto` might not solve goals that `tauto` would be able to solve (e.g. goals involving universal quantifiers).

8.12.6 `firstorder`

The tactic `firstorder` is an *experimental* extension of `tauto` to first-order reasoning, written by Pierre Corbineau. It is not restricted to usual logical connectives but instead may reason about any first-order class inductive definition.

Variants:

1. `firstorder` *tactic*

   Tries to solve the goal with *tactic* when no logical rule may apply.

2. `firstorder with` $ident_1$ `...` $ident_n$

   Adds lemmas $ident_1$ ... $ident_n$ to the proof-search environment.

3. `firstorder using` $ident_1$ `...` $ident_n$

   Adds lemmas in `auto` hints bases $ident_1$ ... $ident_n$ to the proof-search environment.

Proof-search is bounded by a depth parameter which can be set by typing the `Set Firstorder Depth` n vernacular command.

8.12.7 `congruence`

The tactic `congruence`, by Pierre Corbineau, implements the standard Nelson and Oppen congruence closure algorithm, which is a decision procedure for ground equalities with uninterpreted symbols. It also include the constructor theory (see 8.9.4 and 8.9.3). If the goal is a non-quantified equality, `congruence` tries to prove it with non-quantified equalities in the context. Otherwise it tries to infer a discriminable equality from those in the context. Alternatively, congruence tries to prove that a hypothesis is equal to the goal or to the negation of another hypothesis.

`congruence` is also able to take advantage of hypotheses stating quantified equalities, you have to provide a bound for the number of extra equalities generated that way. Please note that one of the members of the equality must contain all the quantified variables in order for `congruence` to match against it.

```
Coq < Theorem T:
Coq <   a=(f a) -> (g b (f a))=(f (f a)) -> (g a b)=(f (g b a)) -> (g a b)=a.
1 subgoal

  ============================
   a = f a -> g b (f a) = f (f a) -> g a b = f (g b a) -> g a b = a

Coq < intros.
1 subgoal

  H : a = f a
  H0 : g b (f a) = f (f a)
  H1 : g a b = f (g b a)
  ============================
   g a b = a

Coq < congruence.
Proof completed.
```

```
Coq < Theorem inj : f = pair a -> Some (f c) = Some (f d) -> c=d.
1 subgoal

  ============================
   f = pair a -> Some (f c) = Some (f d) -> c = d

Coq < intros.
1 subgoal

  H : f = pair a
  H0 : Some (f c) = Some (f d)
  ============================
   c = d

Coq < congruence.
Proof completed.
```

Variants:

1. `congruence` *n*
   Tries to add at most *n* instances of hypotheses stating quantified equalities to the problem in order to solve it. A bigger value of *n* does not make success slower, only failure. You might consider adding some lemmas as hypotheses using `assert` in order for congruence to use them.

Variants:

1. `congruence with` $term_1$ `...` $term_n$
   Adds $term_1$ `...` $term_n$ to the pool of terms used by `congruence`. This helps in case you have partially applied constructors in your goal.

Error messages:

1. `I don't know how to handle dependent equality`
   The decision procedure managed to find a proof of the goal or of a discriminable equality but this proof couldn't be built in Coq because of dependently-typed functions.

2. `I couldn't solve goal`
   The decision procedure didn't find any way to solve the goal.

3. `Goal is solvable by congruence but some arguments are missing. Try "congruence with ...", replacing metavariables by arbitrary terms.`
   The decision procedure could solve the goal with the provision that additional arguments are supplied for some partially applied constructors. Any term of an appropriate type will allow the tactic to successfully solve the goal. Those additional arguments can be given to `congruence` by filling in the holes in the terms given in the error message, using the `with` variant described above.

8.12.8 `omega`

The tactic `omega`, due to Pierre Crégut, is an automatic decision procedure for Presburger arithmetic. It solves quantifier-free formulas built with `~`, `\/`, `/\`, `->` on top of equalities, inequalities and disequalities on both the type `nat` of natural numbers and `Z` of binary integers. This tactic must be loaded by the command `Require Import Omega`. See the additional documentation about `omega` (see Chapter 19).

8.12.9 `ring` and `ring_simplify` *term*$_1$ ... *term*$_n$

The `ring` tactic solves equations upon polynomial expressions of a ring (or semi-ring) structure. It proceeds by normalizing both hand sides of the equation (w.r.t. associativity, commutativity and distributivity, constant propagation) and comparing syntactically the results.

`ring_simplify` applies the normalization procedure described above to the terms given. The tactic then replaces all occurrences of the terms given in the conclusion of the goal by their normal forms. If no term is given, then the conclusion should be an equation and both hand sides are normalized.

See Chapter 23 for more information on the tactic and how to declare new ring structures.

8.12.10 `field`, `field_simplify` *term*$_1$ ... *term*$_n$ and `field_simplify_eq`

The `field` tactic is built on the same ideas as `ring`: this is a reflexive tactic that solves or simplifies equations in a field structure. The main idea is to reduce a field expression (which is an extension of ring expressions with the inverse and division operations) to a fraction made of two polynomial expressions.

Tactic `field` is used to solve subgoals, whereas `field_simplify` *term*$_1$ ... *term*$_n$ replaces the provided terms by their reduced fraction. `field_simplify_eq` applies when the conclusion is an equation: it simplifies both hand sides and multiplies so as to cancel denominators. So it produces an equation without division nor inverse.

All of these 3 tactics may generate a subgoal in order to prove that denominators are different from zero.

See Chapter 23 for more information on the tactic and how to declare new field structures.

Example:

```
Coq < Require Import Reals.

Coq < Goal forall x y:R,
Coq <      (x * y > 0)%R ->
Coq <      (x * (1 / x + x / (x + y)))%R =
Coq <      ((- 1 / y) * y * (- x * (x / (x + y)) - 1))%R.

Coq < intros; field.
1 subgoal

  x : R
  y : R
  H : (x * y > 0)%R
  ============================
   (x + y)%R <> 0%R /\ y <> 0%R /\ x <> 0%R
```

See also: file `contrib/setoid_ring/RealField.v` for an example of instantiation, theory `theories/Reals` for many examples of use of `field`.

8.12.11 `fourier`

This tactic written by Loïc Pottier solves linear inequalities on real numbers using Fourier's method [63]. This tactic must be loaded by `Require Import Fourier`.

Example:

```
Coq < Require Import Reals.

Coq < Require Import Fourier.

Coq < Goal forall x y:R, (x < y)%R -> (y + 1 >= x - 1)%R.

Coq < intros; fourier.
Proof completed.
```

8.12.12 `autorewrite with` *ident*$_1$ `...` *ident*$_n$`.`

This tactic [5] carries out rewritings according the rewriting rule bases *ident*$_1$ `...` *ident*$_n$.

Each rewriting rule of a base *ident*$_i$ is applied to the main subgoal until it fails. Once all the rules have been processed, if the main subgoal has progressed (e.g., if it is distinct from the initial main goal) then the rules of this base are processed again. If the main subgoal has not progressed then the next base is processed. For the bases, the behavior is exactly similar to the processing of the rewriting rules.

The rewriting rule bases are built with the `Hint Rewrite` vernacular command.

Warning: This tactic may loop if you build non terminating rewriting systems.

Variant:

1. `autorewrite with` *ident*$_1$ `...` *ident*$_n$ `using` *tactic*
   Performs, in the same way, all the rewritings of the bases *ident*$_1$ ... *ident*$_n$ applying *tactic* to the main subgoal after each rewriting step.

2. `autorewrite with` *ident*$_1$ `...` *ident*$_n$ `in` *qualid*

   Performs all the rewritings in hypothesis *qualid*.

3. `autorewrite with` *ident*$_1$ `...` *ident*$_n$ `in` *qualid* `using` *tactic*

   Performs all the rewritings in hypothesis *qualid* applying *tactic* to the main subgoal after each rewriting step.

4. `autorewrite with` *ident*$_1$ `...` *ident*$_n$ `in` *clause* Performs all the rewritings in the clause *clause*.
   The *clause* argument must not contain any `type of` nor `value of`.

See also: Section 8.13.4 for feeding the database of lemmas used by `autorewrite`.

See also: Section 10.7 for examples showing the use of this tactic.

[5] The behavior of this tactic has much changed compared to the versions available in the previous distributions (V6). This may cause significant changes in your theories to obtain the same result. As a drawback of the re-engineering of the code, this tactic has also been completely revised to get a very compact and readable version.

8.13 Controlling automation

8.13.1 The hints databases for `auto` and `eauto`

The hints for `auto` and `eauto` are stored in databases. Each database maps head symbols to a list of hints. One can use the command `Print Hint` *ident* to display the hints associated to the head symbol *ident* (see 8.13.3). Each hint has a cost that is an nonnegative integer, and an optional pattern. The hints with lower cost are tried first. A hint is tried by `auto` when the conclusion of the current goal matches its pattern or when it has no pattern.

Creating Hint databases

One can optionally declare a hint database using the command `Create HintDb`. If a hint is added to an unknown database, it will be automatically created.

`Create HintDb` *ident* [`discriminated`]

This command creates a new database named *ident*. The database is implemented by a Discrimination Tree (DT) that serves as an index of all the lemmas. The DT can use transparency information to decide if a constant should be indexed or not (c.f. 8.13.1), making the retrieval more efficient. The legacy implementation (the default one for new databases) uses the DT only on goals without existentials (i.e., auto goals), for non-Immediate hints and do not make use of transparency hints, putting more work on the unification that is run after retrieval (it keeps a list of the lemmas in case the DT is not used). The new implementation enabled by the `discriminated` option makes use of DTs in all cases and takes transparency information into account. However, the order in which hints are retrieved from the DT may differ from the order in which they were inserted, making this implementation observationaly different from the legacy one.

Variants:

1. `Local Hint` *hint_definition* **:** $ident_1 \ldots ident_n$

   This is used to declare a hint database that must not be exported to the other modules that require and import the current module. Inside a section, the option `Local` is useless since hints do not survive anyway to the closure of sections.

The general command to add a hint to some database $ident_1, \ldots, ident_n$ is:

`Hint` *hint_definition* **:** $ident_1 \ldots ident_n$

where *hint_definition* is one of the following expressions:

- `Resolve` *term*

  This command adds `apply` *term* to the hint list with the head symbol of the type of *term*. The cost of that hint is the number of subgoals generated by `apply` *term*.

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

  If the inferred type of *term* contains a dependent quantification on a predicate, it is added to the hint list of `eapply` instead of the hint list of `apply`. In this case, a warning is printed since the

hint is only used by the tactic `eauto` (see 8.12.2). A typical example of a hint that is used only by `eauto` is a transitivity lemma.

Error messages:

1. `Bound head variable`
   The head symbol of the type of *term* is a bound variable such that this tactic cannot be associated to a constant.
2. *term* `cannot be used as a hint`
   The type of *term* 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. `Resolve` $term_1 \ldots term_m$
   Adds each `Resolve` $term_i$.

- `Immediate` *term*

  This command adds `apply` *term*`;` `trivial` to the hint list associated with the head symbol of the type of *ident* in the given database. This tactic will fail if all the subgoals generated by `apply` *term* are not solved immediately by the `trivial` tactic (which only tries tactics with cost 0).

  This command is useful for theorems such as 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 messages:

  1. `Bound head variable`
  2. *term* `cannot be used as a hint`

  Variants:

  1. `Immediate` $term_1 \ldots term_m$
     Adds each `Immediate` $term_i$.

- `Constructors` *ident*

  If *ident* is an inductive type, this command adds all its constructors as hints of type `Resolve`. Then, when the conclusion of current goal has the form `(`*ident* `...)`, `auto` will try to apply each constructor.

  Error messages:

  1. *ident* `is not an inductive type`
  2. *ident* `not declared`

  Variants:

1. `Constructors` *ident*$_1$ ... *ident*$_m$
   Adds each `Constructors` *ident*$_i$.

- `Unfold` *qualid*

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

  Variants:

  1. `Unfold` *ident*$_1$ ... *ident*$_m$
     Adds each `Unfold` *ident*$_i$.

- `Transparent,Opaque` *qualid*

  This adds a transparency hint to the database, making *qualid* a transparent or opaque constant during resolution. This information is used during unification of the goal with any lemma in the database and inside the discrimination network to relax or constrain it in the case of `discriminated` databases.

  Variants:

  1. `Transparent,Opaque` *ident*$_1$ ... *ident*$_m$
     Declares each *ident*$_i$ as a transparent or opaque constant.

- `Extern` *num* `[`*pattern*`]` `=>` *tactic*

  This hint type is to extend `auto` with tactics other than `apply` and `unfold`. For that, we must specify a cost, an optional pattern and a tactic to execute. Here is an example:

```
Hint Extern 4 ~(?=?) => discriminate.
```

  Now, when the head of the goal is a disequality, `auto` will try `discriminate` if it does not manage to solve the goal with hints with a cost less than 4.

  One can even use some sub-patterns of the pattern in the tactic script. A sub-pattern is a question mark followed by an ident, like `?X1` or `?X2`. Here is an example:

```
Coq < Require Import List.
Coq < Hint Extern 5   ({?X1 = ?X2} + {?X1 <> ?X2}) =>
Coq <  generalize X1, X2; decide equality : eqdec.
Coq < Goal
Coq < forall a b:list (nat * nat), {a = b} + {a <> b}.
1 subgoal

  ============================
   forall a b : list (nat * nat), {a = b} + {a <> b}
Coq < info auto with eqdec.
 == intro a; intro b; generalize a, b; decide equality;
    generalize a1, p; decide equality.
    generalize b1, n0; decide equality.
```

```
    generalize a3, n; decide equality.

Proof completed.
```

Remark: One can use an `Extern` hint with no pattern to do pattern-matching on hypotheses using `match goal with` inside the tactic.

Variants:

1. `Hint` *hint_definition*

   No database name is given: the hint is registered in the `core` database.

2. `Hint Local` *hint_definition* `:` $ident_1 \ldots ident_n$

   This is used to declare hints that must not be exported to the other modules that require and import the current module. Inside a section, the option `Local` is useless since hints do not survive anyway to the closure of sections.

3. `Hint Local` *hint_definition*

   Idem for the `core` database.

8.13.2 Hint databases defined in the COQ standard library

Several hint databases are defined in the COQ standard library. The actual content of a database is the collection of the hints declared to belong to this database in each of the various modules currently loaded. Especially, requiring new modules potentially extend a database. At COQ startup, only the `core` and `v62` databases are non empty and can be used.

`core` This special database is automatically used by `auto`. It contains only basic lemmas about negation, conjunction, and so on from. Most of the hints in this database come from the `Init` and `Logic` directories.

`arith` This database contains all lemmas about Peano's arithmetic proved in the directories `Init` and `Arith`

`zarith` contains lemmas about binary signed integers from the directories `theories/ZArith`. When required, the module `Omega` also extends the database `zarith` with a high-cost hint that calls `omega` on equations and inequalities in `nat` or `Z`.

`bool` contains lemmas about booleans, mostly from directory `theories/Bool`.

`datatypes` is for lemmas about lists, streams and so on that are mainly proved in the `Lists` subdirectory.

`sets` contains lemmas about sets and relations from the directories `Sets` and `Relations`.

`typeclass_instances` contains all the type class instances declared in the environment, including those used for `setoid_rewrite`, from the `Classes` directory.

There is also a special database called `v62`. It collects all hints that were declared in the versions of Coq prior to version 6.2.4 when the databases `core`, `arith`, and so on were introduced. The purpose of the database `v62` is to ensure compatibility with further versions of Coq for developments done in versions prior to 6.2.4 (`auto` being replaced by `auto with v62`). The database `v62` is intended not to be extended (!). It is not included in the hint databases list used in the `auto with *` tactic.

Furthermore, you are advised not to put your own hints in the `core` database, but use one or several databases specific to your development.

8.13.3 `Print Hint`

This command displays all hints that apply to the current goal. It fails if no proof is being edited, while the two variants can be used at every moment.

Variants:

1. `Print Hint` *ident*

   This command displays only tactics associated with *ident* in the hints list. This is independent of the goal being edited, so this command will not fail if no goal is being edited.

2. `Print Hint *`

   This command displays all declared hints.

3. `Print HintDb` *ident*

   This command displays all hints from database *ident*.

8.13.4 `Hint Rewrite` $term_1 \ldots term_n$: *ident*

This vernacular command adds the terms $term_1 \ldots term_n$ (their types must be equalities) in the rewriting base *ident* with the default orientation (left to right). Notice that the rewriting bases are distinct from the `auto` hint bases and that `auto` does not take them into account.

This command is synchronous with the section mechanism (see 2.4): when closing a section, all aliases created by `Hint Rewrite` in that section are lost. Conversely, when loading a module, all `Hint Rewrite` declarations at the global level of that module are loaded.

Variants:

1. `Hint Rewrite ->` $term_1 \ldots term_n$: *ident*
   This is strictly equivalent to the command above (we only make explicit the orientation which otherwise defaults to `->`).

2. `Hint Rewrite <-` $term_1 \ldots term_n$: *ident*
   Adds the rewriting rules $term_1 \ldots term_n$ with a right-to-left orientation in the base *ident*.

3. `Hint Rewrite` $term_1 \ldots term_n$ `using` *tactic* : *ident*
   When the rewriting rules $term_1 \ldots term_n$ in *ident* will be used, the tactic *tactic* will be applied to the generated subgoals, the main subgoal excluded.

4. `Print Rewrite HintDb` *ident*

   This command displays all rewrite hints contained in *ident*.

8.13.5 Hints and sections

Hints provided by the `Hint` commands are erased when closing a section. Conversely, all hints of a module `A` that are not defined inside a section (and not defined with option `Local`) become available when the module `A` is imported (using e.g. `Require Import A.`).

8.13.6 Setting implicit automation tactics

`Proof with` *tactic*`.`

This command may be used to start a proof. It defines a default tactic to be used each time a tactic command $tactic_1$ is ended by "`...`". In this case the tactic command typed by the user is equivalent to $tactic_1$;*tactic*.

See also: `Proof.` in Section 7.1.5.

`Declare Implicit Tactic` *tactic*`.`

This command declares a tactic to be used to solve implicit arguments that Coq does not know how to solve by unification. It is used every time the term argument of a tactic has one of its holes not fully resolved.

Here is an example:

```
Coq < Parameter quo : nat -> forall n:nat, n<>0 -> nat.
quo is assumed
Coq < Notation "x // y" := (quo x y _) (at level 40).
Coq <
Coq < Declare Implicit Tactic assumption.
Coq < Goal forall n m, m<>0 -> { q:nat & { r | q * m + r = n } }.
1 subgoal

  ============================
   forall n m : nat, m <> 0 -> {q : nat &  {r : nat | q * m + r = n}}
Coq < intros.
1 subgoal

  n : nat
  m : nat
  H : m <> 0
  ============================
   {q : nat &  {r : nat | q * m + r = n}}
Coq < exists (n // m).
1 subgoal

  n : nat
  m : nat
  H : m <> 0
  ============================
   {r : nat | n // m * m + r = n}
```

The tactic `exists (n // m)` did not fail. The hole was solved by `assumption` so that it behaved as `exists (quo n m H)`.

8.14 Generation of induction principles with `Scheme`

The `Scheme` command is a high-level tool for generating automatically (possibly mutual) induction principles for given types and sorts. Its syntax follows the schema:

`Scheme` *ident*$_1$ `:= Induction for` *ident'*$_1$ `Sort` *sort*$_1$
`with`
  ...
`with` *ident*$_m$ `:= Induction for` *ident'*$_m$ `Sort` *sort*$_m$

where *ident'*$_1$ ... *ident'*$_m$ are different inductive type identifiers belonging to the same package of mutual inductive definitions. This command generates *ident*$_1$... *ident*$_m$ to be mutually recursive definitions. Each term *ident*$_i$ proves a general principle of mutual induction for objects in type *term*$_i$.

Variants:

1. `Scheme` *ident*$_1$ `:= Minimality for` *ident'*$_1$ `Sort` *sort*$_1$
   `with`
     ...
   `with` *ident*$_m$ `:= Minimality for` *ident'*$_m$ `Sort` *sort*$_m$

   Same as before but defines a non-dependent elimination principle more natural in case of inductively defined relations.

2. `Scheme Equality for` *ident*$_1$

   Tries to generate a boolean equality and a proof of the decidability of the usual equality.

3. `Scheme Induction for` *ident*$_1$ `Sort` *sort*$_1$
   `with`
     ...
   `with Induction for` *ident*$_m$ `Sort` *sort*$_m$

   If you do not provide the name of the schemes, they will be automatically computed from the sorts involved (works also with Minimality).

See also: Section 10.3

8.14.1 Automatic declaration of schemes

It is possible to deactivate the automatic declaration of the induction principles when defining a new inductive type with the `UnSet Elimination Schemes` command. It may be reactivated at any time with `Set Elimination Schemes`.

You can also activate the automatic declaration of those boolean equalities (see the second variant of `Scheme`) with the `Set Equality Scheme` command. However you have to be careful with this option since Coq may now reject well-defined inductive types because it cannot compute a boolean equality for them.

8.14.2 Combined Scheme

The Combined Scheme command is a tool for combining induction principles generated by the Scheme command. Its syntax follows the schema :
Combined Scheme *ident*$_0$ from *ident*$_1$, .., *ident*$_n$
ident$_1$ ... *ident*$_n$ are different inductive principles that must belong to the same package of mutual inductive principle definitions. This command generates *ident*$_0$ to be the conjunction of the principles: it is built from the common premises of the principles and concluded by the conjunction of their conclusions.

See also: Section 10.3.1

8.15 Generation of induction principles with Functional Scheme

The Functional Scheme command is a high-level experimental tool for generating automatically induction principles corresponding to (possibly mutually recursive) functions. Its syntax follows the schema:

```
Functional Scheme ident_1 := Induction for ident'_1 Sort sort_1
with
  ...
with ident_m := Induction for ident'_m Sort sort_m
```

where *ident*'$_1$ ... *ident*'$_m$ are different mutually defined function names (they must be in the same order as when they were defined). This command generates the induction principles *ident*$_1$... *ident*$_m$, following the recursive structure and case analyses of the functions *ident*'$_1$ ... *ident*'$_m$.

Functional Scheme There is a difference between obtaining an induction scheme by using Functional Scheme on a function defined by Function or not. Indeed Function generally produces smaller principles, closer to the definition written by the user.

See also: Section 10.4

8.16 Simple tactic macros

A simple example has more value than a long explanation:

```
Coq < Ltac Solve := simpl; intros; auto.
Solve is defined

Coq < Ltac ElimBoolRewrite b H1 H2 :=
Coq <   elim b; [ intros; rewrite H1; eauto | intros; rewrite H2; eauto ].
ElimBoolRewrite is defined
```

The tactics macros are synchronous with the Coq section mechanism: a tactic definition is deleted from the current environment when you close the section (see also 2.4) where it was defined. If you want that a tactic macro defined in a module is usable in the modules that require it, you should put it outside of any section.

Chapter 9 gives examples of more complex user-defined tactics.

Chapter 9

The tactic language

This chapter gives a compact documentation of Ltac, the tactic language available in Coq. We start by giving the syntax, and next, we present the informal semantics. If you want to know more regarding this language and especially about its foundations, you can refer to [41]. Chapter 10 is devoted to giving examples of use of this language on small but also with non-trivial problems.

9.1 Syntax

The syntax of the tactic language is given Figures 9.1 and 9.2. See Chapter 1 for a description of the BNF metasyntax used in these grammar rules. Various already defined entries will be used in this chapter: entries *natural*, *integer*, *ident*, *qualid*, *term*, *cpattern* and *atomic_tactic* represent respectively the natural and integer numbers, the authorized identificators and qualified names, Coq's terms and patterns and all the atomic tactics described in Chapter 8. The syntax of *cpattern* is the same as that of terms, but it is extended with pattern matching metavariables. In *cpattern*, a pattern-matching metavariable is represented with the syntax `?id` where `id` is an *ident*. The notation _ can also be used to denote metavariable whose instance is irrelevant. In the notation `?id`, the identifier allows us to keep instantiations and to make constraints whereas _ shows that we are not interested in what will be matched. On the right hand side of pattern-matching clauses, the named metavariable are used without the question mark prefix. There is also a special notation for second-order pattern-matching problems: in an applicative pattern of the form `@?id` $\texttt{id}_1$ `...` $\texttt{id}_n$, the variable `id` matches any complex expression with (possible) dependencies in the variables $\texttt{id}_1$ `...` $\texttt{id}_n$ and returns a functional term of the form `fun` $\texttt{id}_1$ `...` $\texttt{id}_n$ `=>` *term*.

The main entry of the grammar is *expr*. This language is used in proof mode but it can also be used in toplevel definitions as shown in Figure 9.3.

Remarks:

1. The infix tacticals "... `||` ..." and "... `;` ..." are associative.
2. In *tacarg*, there is an overlap between *qualid* as a direct tactic argument and *qualid* as a particular case of *term*. The resolution is done by first looking for a reference of the tactic language and if it fails, for a reference to a term. To force the resolution as a reference of the tactic language, use the form `ltac :` *qualid*. To force the resolution as a reference to a term, use the syntax `(`*qualid*`)`.
3. As shown by the figure, tactical `||` binds more than the prefix tacticals `try`, `repeat`, `do`, `info` and `abstract` which themselves bind more than the postfix tactical "`... ;[ ... ]`" which binds more than "... `;` ...".

For instance

`try repeat` *tactic*$_1$ `||` *tactic*$_2$`;` *tactic*$_3$`; [` *tactic*$_{31}$ `| ... |` *tactic*$_{3n}$ `];` *tactic*$_4$`.`

is understood as

`(try (repeat (`*tactic*$_1$ `||` *tactic*$_2$`)));`
`((`*tactic*$_3$`; [` *tactic*$_{31}$ `| ... |` *tactic*$_{3n}$ `]);` *tactic*$_4$`).`

9.2 Semantics

Tactic expressions can only be applied in the context of a goal. The evaluation yields either a term, an integer or a tactic. Intermediary results can be terms or integers but the final result must be a tactic which is then applied to the current goal.

There is a special case for `match goal` expressions of which the clauses evaluate to tactics. Such expressions can only be used as end result of a tactic expression (never as argument of a non recursive local definition or of an application).

The rest of this section explains the semantics of every construction of Ltac.

Sequence

A sequence is an expression of the following form:

expr$_1$ `;` *expr*$_2$

expr$_1$ and *expr*$_2$ are evaluated to v_1 and v_2. v_1 and v_2 must be tactic values. v_1 is then applied and v_2 is applied to every subgoal generated by the application of v_1. Sequence is left-associative.

General sequence

We can generalize the previous sequence operator as

expr$_0$ `; [` *expr*$_1$ `| ... |` *expr*$_n$ `]`

expr$_i$ is evaluated to v_i, for $i = 0, ..., n$. v_0 is applied and v_i is applied to the i-th generated subgoal by the application of v_0, for $= 1, ..., n$. It fails if the application of v_0 does not generate exactly n subgoals.

Variant: If no tactic is given for the i-th generated subgoal, it behaves as if the tactic `idtac` were given. For instance, `split ; [ | auto ]` is a shortcut for `split ; [ idtac | auto ]`.

For loop

There is a for loop that repeats a tactic *num* times:

`do` *num expr*

expr is evaluated to v. v must be a tactic value. v is applied *num* times. Supposing *num* > 1, after the first application of v, v is applied, at least once, to the generated subgoals and so on. It fails if the application of v fails before the *num* applications have been completed.

```
expr          ::= expr ; expr
                | expr ; [ expr | ... | expr ]
                | tacexpr3

tacexpr3      ::= do (natural | ident) tacexpr3
                | info tacexpr3
                | progress tacexpr3
                | repeat tacexpr3
                | try tacexpr3
                | tacexpr2

tacexpr2      ::= tacexpr1 || tacexpr3
                | tacexpr1

tacexpr1      ::= fun name ... name => atom
                | let [rec] let_clause with ... with let_clause in atom
                | match goal with context_rule | ... | context_rule end
                | match reverse goal with context_rule | ... | context_rule end
                | match expr with match_rule | ... | match_rule end
                | lazymatch goal with context_rule | ... | context_rule end
                | lazymatch reverse goal with context_rule | ... | context_rule end
                | lazymatch expr with match_rule | ... | match_rule end
                | abstract atom
                | abstract atom using ident
                | first [ expr | ... | expr ]
                | solve [ expr | ... | expr ]
                | idtac [message_token ... message_token]
                | fail [natural] [message_token ... message_token]
                | fresh | fresh string
                | context ident [ term ]
                | eval redexpr in term
                | type of term
                | external string string tacarg ... tacarg
                | constr : term
                | atomic_tactic
                | qualid tacarg ... tacarg
                | atom

atom          ::= qualid
                | ()
                | integer
                | ( expr )

message_token ::= string | term | integer
```

Figure 9.1: Syntax of the tactic language

```
tacarg          ::= qualid
                  | ()
                  | ltac : atom
                  | term

let_clause      ::= ident [name ... name] := expr

context_rule    ::= context_hyps , ... , context_hyps |- cpattern => expr
                  | |- cpattern => expr
                  | _ => expr

context_hyps    ::= name : cpattern

match_rule      ::= cpattern => expr
                  | context [ident] [ cpattern ] => expr
                  | _ => expr
```

Figure 9.2: Syntax of the tactic language (continued)

```
top             ::= Ltac ltac_def with ... with ltac_def

ltac_def        ::= ident [ident ... ident] := expr
                  | qualid [ident ... ident] ::=expr
```

Figure 9.3: Tactic toplevel definitions

Repeat loop

We have a repeat loop with:

`repeat` *expr*

expr is evaluated to v. If v denotes a tactic, this tactic is applied to the goal. If the application fails, the tactic is applied recursively to all the generated subgoals until it eventually fails. The recursion stops in a subgoal when the tactic has failed. The tactic `repeat` *expr* itself never fails.

Error catching

We can catch the tactic errors with:

`try` *expr*

expr is evaluated to v. v must be a tactic value. v is applied. If the application of v fails, it catches the error and leaves the goal unchanged. If the level of the exception is positive, then the exception is re-raised with its level decremented.

Detecting progress

We can check if a tactic made progress with:

`progress` *expr*

expr is evaluated to v. v must be a tactic value. v is applied. If the application of v produced one subgoal equal to the initial goal (up to syntactical equality), then an error of level 0 is raised.

Error message: `Failed to progress`

Branching

We can easily branch with the following structure:

expr$_1$ `||` *expr*$_2$

expr$_1$ and *expr*$_2$ are evaluated to v_1 and v_2. v_1 and v_2 must be tactic values. v_1 is applied and if it fails to progress then v_2 is applied. Branching is left-associative.

First tactic to work

We may consider the first tactic to work (i.e. which does not fail) among a panel of tactics:

`first [` *expr*$_1$ `|` ... `|` *expr*$_n$ `]`

expr$_i$ are evaluated to v_i and v_i must be tactic values, for $i = 1, ..., n$. Supposing $n > 1$, it applies v_1, if it works, it stops else it tries to apply v_2 and so on. It fails when there is no applicable tactic.

Error message: `No applicable tactic`

Solving

We may consider the first to solve (i.e. which generates no subgoal) among a panel of tactics:

`solve [` *expr*$_1$ `|` ... `|` *expr*$_n$ `]`

expr$_i$ are evaluated to v_i and v_i must be tactic values, for $i = 1, ..., n$. Supposing $n > 1$, it applies v_1, if it solves, it stops else it tries to apply v_2 and so on. It fails if there is no solving tactic.

Error message: `Cannot solve the goal`

Identity

The constant `idtac` is the identity tactic: it leaves any goal unchanged but it appears in the proof script.

Variant: `idtac` *message_token* ... *message_token*

This prints the given tokens. Strings and integers are printed literally. If a term is given, it is printed, its variables being interpreted in the current environment. In particular, if a variable is given, its value is printed.

Failing

The tactic `fail` is the always-failing tactic: it does not solve any goal. It is useful for defining other tacticals since it can be catched by `try` or `match goal`.

Variants:

1. `fail` n
   The number n is the failure level. If no level is specified, it defaults to 0. The level is used by `try` and `match goal`. If 0, it makes `match goal` considering the next clause (backtracking). If non zero, the current `match goal` block or `try` command is aborted and the level is decremented.

2. `fail` *message_token* ... *message_token*
   The given tokens are used for printing the failure message.

3. `fail` n *message_token* ... *message_token*
   This is a combination of the previous variants.

Error message: `Tactic Failure` *message* `(level` n`).`

Local definitions

Local definitions can be done as follows:

`let` *ident*$_1$ `:=` *expr*$_1$
`with` *ident*$_2$ `:=` *expr*$_2$
...
`with` *ident*$_n$ `:=` *expr*$_n$ `in`
expr

each *expr*$_i$ is evaluated to v_i, then, *expr* is evaluated by substituting v_i to each occurrence of *ident*$_i$, for $i = 1, ..., n$. There is no dependencies between the *expr*$_i$ and the *ident*$_i$.

Local definitions can be recursive by using `let rec` instead of `let`. In this latter case, the definitions are evaluated lazily so that the `rec` keyword can be used also in non recursive cases so as to avoid the eager evaluation of local definitions.

Application

An application is an expression of the following form:

qualid *tacarg*$_1$ ... *tacarg*$_n$

The reference *qualid* must be bound to some defined tactic definition expecting at least n arguments. The expressions *expr*$_i$ are evaluated to v_i, for $i = 1, ..., n$.

Function construction

A parameterized tactic can be built anonymously (without resorting to local definitions) with:

`fun` *ident*$_1$ ... *ident*$_n$ `=>` *expr*

Indeed, local definitions of functions are a syntactic sugar for binding a `fun` tactic to an identifier.

Pattern matching on terms

We can carry out pattern matching on terms with:

```
match expr with
cpattern_1 => expr_1
| cpattern_2 => expr_2
...
| cpattern_n => expr_n
| _ => expr_{n+1}
end
```

The expression *expr* is evaluated and should yield a term which is matched against $cpattern_1$. The matching is non-linear: if a metavariable occurs more than once, it should match the same expression every time. It is first-order except on the variables of the form `@?id` that occur in head position of an application. For these variables, the matching is second-order and returns a functional term.

If the matching with $cpattern_1$ succeeds, then $expr_1$ is evaluated into some value by substituting the pattern matching instantiations to the metavariables. If $expr_1$ evaluates to a tactic and the `match` expression is in position to be applied to a goal (e.g. it is not bound to a variable by a `let in`), then this tactic is applied. If the tactic succeeds, the list of resulting subgoals is the result of the `match` expression. If $expr_1$ does not evaluate to a tactic or if the `match` expression is not in position to be applied to a goal, then the result of the evaluation of $expr_1$ is the result of the `match` expression.

If the matching with $cpattern_1$ fails, or if it succeeds but the evaluation of $expr_1$ fails, or if the evaluation of $expr_1$ succeeds but returns a tactic in execution position whose execution fails, then $cpattern_2$ is used and so on. The pattern _ matches any term and shunts all remaining patterns if any. If all clauses fail (in particular, there is no pattern _) then a no-matching-clause error is raised.

Error messages:

1. `No matching clauses for match`

   No pattern can be used and, in particular, there is no _ pattern.

2. `Argument of match does not evaluate to a term`

   This happens when *expr* does not denote a term.

Variants:

1. There is a special form of patterns to match a subterm against the pattern:

   `context` *ident* `[` *cpattern* `]`

   It matches any term which one subterm matches *cpattern*. If there is a match, the optional *ident* is assign the "matched context", that is the initial term where the matched subterm is replaced by a hole. The definition of `context` in expressions below will show how to use such term contexts.

   If the evaluation of the right-hand-side of a valid match fails, the next matching subterm is tried. If no further subterm matches, the next clause is tried. Matching subterms are considered top-bottom and from left to right (with respect to the raw printing obtained by setting option `Printing All`, see Section 2.9).

```
Coq < Ltac f x :=
Coq <   match x with
Coq <     context f [S ?X] =>
Coq <     idtac X;                      (* To display the evaluation order *)
Coq <     assert (p := refl_equal 1 : X=1);    (* To filter the case X=1 *)
Coq <     let x:= context f[O] in assert (x=O) (* To observe the context *)
Coq <   end.
f is defined

Coq < Goal True.
1 subgoal

  ============================
   True

Coq < f (3+4).
2
1
2 subgoals

  p : 1 = 1
  ============================
   1 + 4 = 0
subgoal 2 is:
 True
```

2. Using `lazymatch` instead of `match` has an effect if the right-hand-side of a clause returns a tactic. With `match`, the tactic is applied to the current goal (and the next clause is tried if it fails). With `lazymatch`, the tactic is directly returned as the result of the whole `lazymatch` block without being first tried to be applied to the goal. Typically, if the `lazymatch` block is bound to some variable x in a `let in`, then tactic expression gets bound to the variable x.

Pattern matching on goals

We can make pattern matching on goals using the following expression:

```
match goal with
  | hyp_{1,1}, ..., hyp_{1,m_1}  |- cpattern_1 => expr_1
  | hyp_{2,1}, ..., hyp_{2,m_2}  |- cpattern_2 => expr_2
 ...
  | hyp_{n,1}, ..., hyp_{n,m_n}  |- cpattern_n => expr_n
  |_   => expr_{n+1}
end
```

If each hypothesis pattern $hyp_{1,i}$, with $i = 1, ..., m_1$ is matched (non-linear first order unification) by an hypothesis of the goal and if *cpattern*$_1$ is matched by the conclusion of the goal, then *expr*$_1$ is evaluated to v_1 by substituting the pattern matching to the metavariables and the real hypothesis names bound to the possible hypothesis names occurring in the hypothesis patterns. If v_1 is a tactic value, then it is applied to the goal. If this application fails, then another combination of hypotheses is tried with the same proof context pattern. If there is no other combination of hypotheses then the second proof context pattern is tried and so on. If the next to last proof context pattern fails then *expr*$_{n+1}$ is evaluated to v_{n+1}

and v_{n+1} is applied. Note also that matching against subterms (using the `context` *ident* `[` *cpattern* `]`) is available and may itself induce extra backtrackings.

Error message: `No matching clauses for match goal`
No clause succeeds, i.e. all matching patterns, if any, fail at the application of the right-hand-side.

It is important to know that each hypothesis of the goal can be matched by at most one hypothesis pattern. The order of matching is the following: hypothesis patterns are examined from the right to the left (i.e. hyp_{i,m_i} before $hyp_{i,1}$). For each hypothesis pattern, the goal hypothesis are matched in order (fresher hypothesis first), but it possible to reverse this order (older first) with the `match reverse goal with` variant.

Variant: Using `lazymatch` instead of `match` has an effect if the right-hand-side of a clause returns a tactic. With `match`, the tactic is applied to the current goal (and the next clause is tried if it fails). With `lazymatch`, the tactic is directly returned as the result of the whole `lazymatch` block without being first tried to be applied to the goal. Typically, if the `lazymatch` block is bound to some variable x in a `let in`, then tactic expression gets bound to the variable x.

```
Coq < Ltac test_lazy :=
Coq <   lazymatch goal with
Coq <   | _ => idtac "here"; fail
Coq <   | _ => idtac "wasn't lazy"; trivial
Coq <   end.
test_lazy is defined

Coq < Ltac test_eager :=
Coq <   match goal with
Coq <   | _ => idtac "here"; fail
Coq <   | _ => idtac "wasn't lazy"; trivial
Coq <   end.
test_eager is defined

Coq < Goal True.
1 subgoal

  ============================
   True

Coq < test_lazy || idtac "was lazy".
here
was lazy
1 subgoal

  ============================
   True

Coq < test_eager || idtac "was lazy".
here
wasn't lazy
Proof completed.
```

Filling a term context

The following expression is not a tactic in the sense that it does not produce subgoals but generates a term to be used in tactic expressions:

`context` *ident* `[` *expr* `]`

ident must denote a context variable bound by a `context` pattern of a `match` expression. This expression evaluates replaces the hole of the value of *ident* by the value of *expr*.

Error message: `not a context variable`

Generating fresh hypothesis names

Tactics sometimes have to generate new names for hypothesis. Letting the system decide a name with the `intro` tactic is not so good since it is very awkward to retrieve the name the system gave. The following expression returns an identifier:

`fresh` *component* `...` *component*

It evaluates to an identifier unbound in the goal. This fresh identifier is obtained by concatenating the value of the *component*'s (each of them is, either an *ident* which has to refer to a name, or directly a name denoted by a *string*). If the resulting name is already used, it is padded with a number so that it becomes fresh. If no component is given, the name is a fresh derivative of the name `H`.

Computing in a constr

Evaluation of a term can be performed with:

`eval` *redexpr* `in` *term*

where *redexpr* is a reduction tactic among `red`, `hnf`, `compute`, `simpl`, `cbv`, `lazy`, `unfold`, `fold`, `pattern`.

Type-checking a term

The following returns the type of *term*:

`type of` *term*

Accessing tactic decomposition

Tactical "`info` *expr*" is not really a tactical. For elementary tactics, this is equivalent to *expr*. For complex tactic like `auto`, it displays the operations performed by the tactic.

Proving a subgoal as a separate lemma

From the outside "`abstract` *expr*" is the same as `solve` *expr*. Internally it saves an auxiliary lemma called *ident*`_subproof`*n* where *ident* is the name of the current goal and *n* is chosen so that this is a fresh name.

This tactical is useful with tactics such as `omega` or `discriminate` that generate huge proof terms. With that tool the user can avoid the explosion at time of the `Save` command without having to cut manually the proof in smaller lemmas.

Variants:

1. `abstract` *expr* `using` *ident*.
   Give explicitly the name of the auxiliary lemma.

Error message: `Proof is not complete`

Calling an external tactic

The tactic `external` allows to run an executable outside the Coq executable. The communication is done via an XML encoding of constructions. The syntax of the command is

`external` "*command*" "*request*" *tacarg* ... *tacarg*

The string *command*, to be interpreted in the default execution path of the operating system, is the name of the external command. The string *request* is the name of a request to be sent to the external command. Finally the list of tactic arguments have to evaluate to terms. An XML tree of the following form is sent to the standard input of the external command.

`<REQUEST req="`*request*`">`
the XML tree of the first argument
...
the XML tree of the last argument
`</REQUEST>`

Conversely, the external command must send on its standard output an XML tree of the following forms:

`<TERM>`
the XML tree of a term
`</TERM>`

or

`<CALL uri="`*ltac_qualified_ident*`">`
the XML tree of a first argument
...
the XML tree of a last argument
`</CALL>`

where *ltac_qualified_ident* is the name of a defined $\mathcal{L}_{tac}$ function and each subsequent XML tree is recursively a `CALL` or a `TERM` node.

The Document Type Definition (DTD) for terms of the Calculus of Inductive Constructions is the one developed as part of the MoWGLI European project. It can be found in the file `dev/doc/cic.dtd` of the Coq source archive.

An example of parser for this DTD, written in the Objective Caml - Camlp4 language, can be found in the file `parsing/g_xml.ml4` of the Coq source archive.

9.3 Tactic toplevel definitions

9.3.1 Defining $\mathcal{L}_{tac}$ functions

Basically, $\mathcal{L}_{tac}$ toplevel definitions are made as follows:

`Ltac` *ident* $ident_1$ ... $ident_n$ `:=` *expr*

This defines a new $\mathcal{L}_{tac}$ function that can be used in any tactic script or new $\mathcal{L}_{tac}$ toplevel definition.

Remark: The preceding definition can equivalently be written:

`Ltac` *ident* `:=` `fun` $ident_1$ ... $ident_n$ `=>` *expr*

Recursive and mutual recursive function definitions are also possible with the syntax:

`Ltac` *ident*$_1$ *ident*$_{1,1}$ ... *ident*$_{1,m_1}$ `:=` *expr*$_1$
`with` *ident*$_2$ *ident*$_{2,1}$ ... *ident*$_{2,m_2}$ `:=` *expr*$_2$
...
`with` *ident*$_n$ *ident*$_{n,1}$ ... *ident*$_{n,m_n}$ `:=` *expr*$_n$

It is also possible to *redefine* an existing user-defined tactic using the syntax:

`Ltac` *qualid ident*$_1$ ... *ident*$_n$ `::=` *expr*

A previous definition of *qualid* must exist in the environment. The new definition will always be used instead of the old one and it goes accross module boundaries.

9.3.2 Printing $\mathcal{L}_{tac}$ tactics

Defined $\mathcal{L}_{tac}$ functions can be displayed using the command

`Print Ltac` *qualid* `.`

9.4 Debugging $\mathcal{L}_{tac}$ tactics

The $\mathcal{L}_{tac}$ interpreter comes with a step-by-step debugger. The debugger can be activated using the command

```
Set Ltac Debug.
```

and deactivated using the command

```
Unset Ltac Debug.
```

To know if the debugger is on, use the command `Test Ltac Debug`. When the debugger is activated, it stops at every step of the evaluation of the current $\mathcal{L}_{tac}$ expression and it prints information on what it is doing. The debugger stops, prompting for a command which can be one of the following:

simple newline:	go to the next step
h:	get help
x:	exit current evaluation
s:	continue current evaluation without stopping
rn:	advance n steps further

Chapter 10

Detailed examples of tactics

This chapter presents detailed examples of certain tactics, to illustrate their behavior.

10.1 `refine`

This tactic applies to any goal. It behaves like `exact` with a big difference : the user can leave some holes (denoted by _ or `(_:`*type*`)`) in the term. `refine` will generate as many subgoals as they are holes in the term. The type of holes must be either synthesized by the system or declared by an explicit cast like `(\_:nat->Prop)`. This low-level tactic can be useful to advanced users.

Example:

```
Coq < Inductive Option : Set :=
Coq <   | Fail : Option
Coq <   | Ok : bool -> Option.

Coq < Definition get : forall x:Option, x <> Fail -> bool.
1 subgoal

  ============================
   forall x : Option, x <> Fail -> bool

Coq < refine
Coq <  (fun x:Option =>
Coq <     match x return x <> Fail -> bool with
Coq <     | Fail => _
Coq <     | Ok b => fun _ => b
Coq <     end).
1 subgoal

  x : Option
  ============================
   Fail <> Fail -> bool

Coq < intros; absurd (Fail = Fail); trivial.
Proof completed.

Coq < Defined.
```

10.2 eapply

Example: Assume we have a relation on `nat` which is transitive:

```
Coq < Variable R : nat -> nat -> Prop.
Coq < Hypothesis Rtrans : forall x y z:nat, R x y -> R y z -> R x z.
Coq < Variables n m p : nat.
Coq < Hypothesis Rnm : R n m.
Coq < Hypothesis Rmp : R m p.
```

Consider the goal `(R n p)` provable using the transitivity of `R`:

```
Coq < Goal R n p.
```

The direct application of `Rtrans` with `apply` fails because no value for `y` in `Rtrans` is found by `apply`:

```
Coq < apply Rtrans.
Unnamed_thm < Unnamed_thm < Toplevel input, characters 144-156:
> apply Rtrans.
> ^^^^^^^^^^^^
Error: Unable to find an instance for the variable y.
```

A solution is to rather apply `(Rtrans n m p)`.

```
Coq < apply (Rtrans n m p).
2 subgoals

  ============================
   R n m
subgoal 2 is:
 R m p
```

More elegantly, `apply Rtrans with (y:=m)` allows to only mention the unknown `m`:

```
Coq <
Coq <   apply Rtrans with (y := m).
2 subgoals

  ============================
   R n m
subgoal 2 is:
 R m p
```

Another solution is to mention the proof of `(R x y)` in `Rtrans`...

```
Coq <
Coq <   apply Rtrans with (1 := Rnm).
1 subgoal

  ============================
   R m p
```

... or the proof of (R y z):

```
Coq <
Coq <   apply Rtrans with (2 := Rmp).
1 subgoal

  ============================
   R n m
```

On the opposite, one can use eapply which postpone the problem of finding m. Then one can apply the hypotheses Rnm and Rmp. This instantiates the existential variable and completes the proof.

```
Coq < eapply Rtrans.
2 subgoals

  ============================
   R n ?9
subgoal 2 is:
 R ?9 p
Coq < apply Rnm.
1 subgoal

  ============================
   R m p
Coq < apply Rmp.
Proof completed.
```

10.3 Scheme

Example 1: *Induction scheme for* tree *and* forest

The definition of principle of mutual induction for tree and forest over the sort Set is defined by the command:

```
Coq < Inductive tree : Set :=
Coq <     node : A -> forest -> tree
Coq < with forest : Set :=
Coq <   | leaf : B -> forest
Coq <   | cons : tree -> forest -> forest.
Coq <
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.
tree_forest_rec
     : forall (P : tree -> Set) (P0 : forest -> Set),
       (forall (a : A) (f : forest), P0 f -> P (node a f)) ->
       (forall b : B, P0 (leaf b)) ->
       (forall t : tree, P t -> forall f1 : forest, P0 f1 -> P0 (cons t f1)) ->
       forall t : tree, P t
```

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

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

```
Coq < Check forest_tree_rec.
forest_tree_rec
     : forall (P : tree -> Set) (P0 : forest -> Set),
       (forall (a : A) (f : forest), P0 f -> P (node a f)) ->
       (forall b : B, P0 (leaf b)) ->
       (forall t : tree, P t -> forall f1 : forest, P0 f1 -> P0 (cons t f1)) ->
       forall f2 : forest, P0 f2
```

Example 2: *Predicates* `odd` *and* `even` *on naturals*

Let `odd` and `even` be inductively defined as:

```
Coq < Inductive odd : nat -> Prop :=
Coq <     oddS : forall n:nat, even n -> odd (S n)
Coq < with even : nat -> Prop :=
Coq <   | evenO : even 0
Coq <   | evenS : forall n:nat, odd n -> even (S n).
```

The 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.
odd_even, even_odd are recursively defined
```

The type of `odd_even` for instance will be:

```
Coq < Check odd_even.
odd_even
     : forall P P0 : nat -> Prop,
       (forall n : nat, even n -> P0 n -> P (S n)) ->
       P0 0 ->
       (forall n : nat, odd n -> P n -> P0 (S n)) ->
       forall n : nat, odd n -> P n
```

The type of `even_odd` shares the same premises but the conclusion is `(n:nat)(even n)->(Q n)`.

10.3.1 `Combined Scheme`

We can define the induction principles for trees and forests using:

```
Coq < Scheme tree_forest_ind := Induction for tree Sort Prop
Coq <   with forest_tree_ind := Induction for forest Sort Prop.
tree_forest_ind, forest_tree_ind are recursively defined
```

Then we can build the combined induction principle which gives the conjunction of the conclusions of each individual principle:

```
Coq < Combined Scheme tree_forest_mutind from tree_forest_ind, forest_tree_ind.
tree_forest_mutind is recursively defined
```

The type of tree_forest_mutrec will be:

```
Coq < Check tree_forest_mutind.
tree_forest_mutind
     : forall (P : tree -> Prop) (P0 : forest -> Prop),
       (forall (a : A) (f : forest), P0 f -> P (node a f)) ->
       (forall b : B, P0 (leaf b)) ->
       (forall t : tree, P t -> forall f1 : forest, P0 f1 -> P0 (cons t f1)) ->
       (forall t : tree, P t) /\ (forall f2 : forest, P0 f2)
```

10.4 Functional Scheme and functional induction

Example 1: *Induction scheme for* div2

We define the function div2 as follows:

```
Coq < Require Import Arith.
Coq < Fixpoint div2 (n:nat) : nat :=
Coq <   match n with
Coq <   | O => 0
Coq <   | S O => 0
Coq <   | S (S n') => S (div2 n')
Coq <   end.
```

The definition of a principle of induction corresponding to the recursive structure of div2 is defined by the command:

```
Coq < Functional Scheme div2_ind := Induction for div2 Sort Prop.
div2_equation is defined
div2_ind is defined
```

You may now look at the type of div2_ind:

```
Coq < Check div2_ind.
div2_ind
     : forall P : nat -> nat -> Prop,
       (forall n : nat, n = 0 -> P 0 0) ->
       (forall n n0 : nat, n = S n0 -> n0 = 0 -> P 1 0) ->
       (forall n n0 : nat,
        n = S n0 ->
        forall n' : nat,
        n0 = S n' -> P n' (div2 n') -> P (S (S n')) (S (div2 n'))) ->
       forall n : nat, P n (div2 n)
```

We can now prove the following lemma using this principle:

```
Coq < Lemma div2_le' : forall n:nat, div2 n <= n.
Coq < intro n.
Coq <  pattern n , (div2 n).
```

```
Coq < apply div2_ind; intros.
3 subgoals

  n : nat
  n0 : nat
  e : n0 = 0
  ============================
   0 <= 0
subgoal 2 is:
 0 <= 1
subgoal 3 is:
 S (div2 n') <= S (S n')
Coq < auto with arith.
Coq < auto with arith.
Coq < simpl; auto with arith.
Coq < Qed.
```

We can use directly the `functional induction` (8.7.7) tactic instead of the pattern/apply trick:

```
Coq < Reset div2_le'.
Coq < Lemma div2_le : forall n:nat, div2 n <= n.
Coq < intro n.
Coq < functional induction (div2 n).
3 subgoals

  ============================
   0 <= 0
subgoal 2 is:
 0 <= 1
subgoal 3 is:
 S (div2 n') <= S (S n')
Coq < auto with arith.
Coq < auto with arith.
Coq < auto with arith.
Coq < Qed.
```

Remark: There is a difference between obtaining an induction scheme for a function by using `Function` (see Section 2.3) and by using `Functional Scheme` after a normal definition using `Fixpoint` or `Definition`. See 2.3 for details.

Example 2: *Induction scheme for* `tree_size`

We define trees by the following mutual inductive type:

```
Coq < Variable A : Set.
Coq < Inductive tree : Set :=
Coq <     node : A -> forest -> tree
Coq < with forest : Set :=
Coq <   | empty : forest
Coq <   | cons : tree -> forest -> forest.
```

We define the function tree_size that computes the size of a tree or a forest. Note that we use Function which generally produces better principles.

```
Coq < Function tree_size (t:tree) : nat :=
Coq <   match t with
Coq <   | node A f => S (forest_size f)
Coq <   end
Coq <  with forest_size (f:forest) : nat :=
Coq <   match f with
Coq <   | empty => 0
Coq <   | cons t f' => (tree_size t + forest_size f')
Coq <   end.
```

Remark: Function generates itself non mutual induction principles tree_size_ind and forest_size_ind:

```
Coq < Check tree_size_ind.
tree_size_ind
     : forall P : tree -> nat -> Prop,
       (forall (t : tree) (A : A) (f : forest),
        t = node A f -> P (node A f) (S (forest_size f))) ->
       forall t : tree, P t (tree_size t)
```

The definition of mutual induction principles following the recursive structure of tree_size and forest_size is defined by the command:

```
Coq < Functional Scheme tree_size_ind2 := Induction for tree_size Sort Prop
Coq < with forest_size_ind2 := Induction for forest_size Sort Prop.
```

You may now look at the type of tree_size_ind2:

```
Coq < Check tree_size_ind2.
tree_size_ind2
     : forall (P : tree -> nat -> Prop) (P0 : forest -> nat -> Prop),
       (forall (t : tree) (A : A) (f : forest),
        t = node A f ->
        P0 f (forest_size f) -> P (node A f) (S (forest_size f))) ->
       (forall f0 : forest, f0 = empty -> P0 empty 0) ->
       (forall (f1 : forest) (t : tree) (f' : forest),
        f1 = cons t f' ->
        P t (tree_size t) ->
        P0 f' (forest_size f') ->
        P0 (cons t f') (tree_size t + forest_size f')) ->
       forall t : tree, P t (tree_size t)
```

10.5 inversion

Generalities about inversion

When working with (co)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.

- we have a 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. tactics for inverting an instance without stocking the inversion lemma in the context; this includes the tactics (dependent) inversion and (dependent) inversion_clear.
2. commands for generating and stocking in the context the inversion lemma corresponding to an instance; this includes Derive (Dependent) Inversion and Derive (Dependent) Inversion_clear.
3. tactics for inverting an instance using an already defined inversion lemma; this includes the tactic inversion ...using.

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.

Example 1: *Non-dependent inversion*

Let's consider the relation Le over natural numbers and the following variables:

```
Coq < Inductive Le : nat -> nat -> Set :=
Coq <   | LeO : forall n:nat, Le 0 n
Coq <   | LeS : forall n m:nat, Le n m -> Le (S n) (S m).
Coq < Variable P : nat -> nat -> Prop.
Coq < Variable Q : forall n m:nat, Le n m -> Prop.
```

For example, consider the goal:

```
Coq < Show.
1 subgoal

  n : nat
  m : nat
  H : Le (S n) m
  ============================
   P n m
```

To prove the goal we may need to reason by cases on H and to derive that m is necessarily of the form $(S\ m_0)$ for certain m_0 and that $(Le\ n\ m_0)$. Deriving these conditions corresponds to prove that the only possible constructor of (Le (S n) m) is LeS and that we can invert the -> in the type of LeS. This inversion is possible because Le is the smallest set closed by the constructors LeO and LeS.

```
Coq < inversion_clear H.
1 subgoal

  n : nat
  m : nat
  m0 : nat
  H0 : Le n m0
  ============================
   P n (S m0)
```

Note that m has been substituted in the goal for (S m0) and that the hypothesis (Le n m0) has been added to the context.

Sometimes it is interesting to have the equality m=(S m0) in the context to use it after. In that case we can use inversion that does not clear the equalities:

```
Coq < Undo.

Coq < inversion H.
1 subgoal

  n : nat
  m : nat
  H : Le (S n) m
  n0 : nat
  m0 : nat
  H1 : Le n m0
  H0 : n0 = n
  H2 : S m0 = m
  ============================
   P n (S m0)
```

Example 2: *Dependent Inversion*

Let us consider the following goal:

```
Coq < Show.
1 subgoal

  n : nat
  m : nat
  H : Le (S n) m
  ============================
   Q (S n) m H
```

As H occurs in the goal, we may want to reason by cases on its structure and so, we would like inversion tactics to substitute H by the corresponding term in constructor form. Neither Inversion nor Inversion_clear make such a substitution. To have such a behavior we use the dependent inversion tactics:

```
Coq < dependent inversion_clear H.
1 subgoal

  n : nat
  m : nat
  m0 : nat
  l : Le n m0
  ============================
   Q (S n) (S m0) (LeS n m0 l)
```

Note that H has been substituted by (LeS n m0 l) and m by (S m0).

Example 3: *using already defined inversion lemmas*

For example, to generate the inversion lemma for the instance (Le (S n) m) and the sort Prop we do:

```
Coq < Derive Inversion_clear leminv with (forall n m:nat, Le (S n) m) Sort
Coq <  Prop.

Coq < Check leminv.
leminv
     : forall (n m : nat) (P : nat -> nat -> Prop),
       (forall m0 : nat, Le n m0 -> P n (S m0)) -> Le (S n) m -> P n m
```

Then we can use the proven inversion lemma:

```
Coq < Show.
1 subgoal

  n : nat
  m : nat
  H : Le (S n) m
  ============================
   P n m

Coq < inversion H using leminv.
1 subgoal

  n : nat
  m : nat
  H : Le (S n) m
  ============================
   forall m0 : nat, Le n m0 -> P n (S m0)
```

10.6 dependent induction

The tactics `dependent induction` and `dependent destruction` are another solution for inverting inductive predicate instances and potentially doing induction at the same time. It is based on the `BasicElim` tactic of Conor McBride which works by abstracting each argument of an inductive instance by a variable and constraining it by equalities afterwards. This way, the usual `induction` and `destruct` tactics can be applied to the abstracted instance and after simplification of the equalities we get the expected goals.

The abstracting tactic is called `generalize_eqs` and it takes as argument an hypothesis to generalize. It uses the `JMeq` datatype defined in `Coq.Logic.JMeq`, hence we need to require it before. For example, revisiting the first example of the inversion documentation above:

```
Coq < Require Import Coq.Logic.JMeq.

Coq < Goal forall n m:nat, Le (S n) m -> P n m.

Coq < intros n m H.

Coq < generalize_eqs H.
1 subgoal

  n : nat
  m : nat
  gen_x : nat
```

```
  H : Le gen_x m
  ============================
   gen_x = S n -> P n m
```

The index `S n` gets abstracted by a variable here, but a corresponding equality is added under the abstract instance so that no information is actually lost. The goal is now almost ammenable to do induction or case analysis. One should indeed first move `n` into the goal to strengthen it before doing induction, or `n` will be fixed in the inductive hypotheses (this does not matter for case analysis). As a rule of thumb, all the variables that appear inside constructors in the indices of the hypothesis should be generalized. This is exactly what the `generalize_eqs_vars` variant does:

```
Coq < generalize_eqs_vars H.
1 subgoal

  m : nat
  gen_x : nat
  H : Le gen_x m
  ============================
   forall n : nat, gen_x = S n -> P n m

Coq < induction H.
2 subgoals

  n : nat
  ============================
   forall n0 : nat, 0 = S n0 -> P n0 n
subgoal 2 is:
 forall n0 : nat, S n = S n0 -> P n0 (S m)
```

As the hypothesis itself did not appear in the goal, we did not need to use an heterogeneous equality to relate the new hypothesis to the old one (which just disappeared here). However, the tactic works just a well in this case, e.g.:

```
Coq < Goal forall n m (p : Le (S n) m), Q (S n) m p.
1 subgoal

  ============================
   forall (n m : nat) (p : Le (S n) m), Q (S n) m p

Coq < intros n m p ; generalize_eqs_vars p.
1 subgoal

  m : nat
  gen_x : nat
  p : Le gen_x m
  ============================
   forall (n : nat) (p0 : Le (S n) m),
   gen_x = S n -> [p : (Le gen_x m)]= [p0 : (Le (S n) m)] -> Q (S n) m p0
```

One drawback of this approach is that in the branches one will have to substitute the equalities back into the instance to get the right assumptions. Sometimes injection of constructors will also be needed to recover the needed equalities. Also, some subgoals should be directly solved because of inconsistent

contexts arising from the constraints on indices. The nice thing is that we can make a tactic based on discriminate, injection and variants of substitution to automatically do such simplifications (which may involve the K axiom). This is what the `simplify_dep_elim` tactic from `Coq.Program.Equality` does. For example, we might simplify the previous goals considerably:

```
Coq < induction p ; simplify_dep_elim.
1 subgoal

  n0 : nat
  m : nat
  p : Le n0 m
  IHp : forall (n : nat) (p0 : Le (S n) m),
        n0 = S n -> [p : (Le n0 m)]= [p0 : (Le (S n) m)] -> Q (S n) m p0
  ============================
   Q (S n0) (S m) (LeS n0 m p)
```

The higher-order tactic `do_depind` defined in `Coq.Program.Equality` takes a tactic and combines the building blocks we've seen with it: generalizing by equalities calling the given tactic with the generalized induction hypothesis as argument and cleaning the subgoals with respect to equalities. Its most important instantiations are `dependent induction` and `dependent destruction` that do induction or simply case analysis on the generalized hypothesis. For example we can redo what we've done manually with `dependent destruction`:

```
Coq < Require Import Coq.Program.Equality.

Coq < Lemma ex : forall n m:nat, Le (S n) m -> P n m.

Coq < intros n m H.

Coq < dependent destruction H.
1 subgoal

  n : nat
  m : nat
  H : Le n m
  ============================
   P n (S m)
```

This gives essentially the same result as inversion. Now if the destructed hypothesis actually appeared in the goal, the tactic would still be able to invert it, contrary to `dependent inversion`. Consider the following example on vectors:

```
Coq < Require Import Coq.Program.Equality.

Coq < Set Implicit Arguments.

Coq < Variable A : Set.

Coq < Inductive vector : nat -> Type :=
Coq < | vnil : vector 0
Coq < | vcons : A -> forall n, vector n -> vector (S n).

Coq < Goal forall n, forall v : vector (S n),
Coq <   exists v' : vector n, exists a : A, v = vcons a v'.

Coq <   intros n v.
```

```
Coq <   dependent destruction v.
1 subgoal

  n : nat
  a : A
  v : vector n
  ============================
   exists v' : vector n, exists a0 : A, vcons a v = vcons a0 v'
```

In this case, the `v` variable can be replaced in the goal by the generalized hypothesis only when it has a type of the form `vector (S n)`, that is only in the second case of the `destruct`. The first one is dismissed because `S n <> 0`.

10.6.1 A larger example

Let's see how the technique works with `induction` on inductive predicates on a real example. We will develop an example application to the theory of simply-typed lambda-calculus formalized in a dependently-typed style:

```
Coq < Inductive type : Type :=
Coq < | base : type
Coq < | arrow : type -> type -> type.

Coq < Notation " t -> t' " := (arrow t t') (at level 20, t' at next level).

Coq < Inductive ctx : Type :=
Coq < | empty : ctx
Coq < | snoc : ctx -> type -> ctx.

Coq < Notation " G , tau " := (snoc G tau) (at level 20, t at next level).

Coq < Fixpoint conc (G D : ctx) : ctx :=
Coq <   match D with
Coq <     | empty => G
Coq <     | snoc D' x => snoc (conc G D') x
Coq <   end.

Coq < Notation " G ; D " := (conc G D) (at level 20).

Coq < Inductive term : ctx -> type -> Type :=
Coq < | ax : forall G tau, term (G, tau) tau
Coq < | weak : forall G tau,
Coq <   term G tau -> forall tau', term (G, tau') tau
Coq < | abs : forall G tau tau',
Coq <   term (G , tau) tau' -> term G (tau -> tau')
Coq < | app : forall G tau tau',
Coq <   term G (tau -> tau') -> term G tau -> term G tau'.
```

We have defined types and contexts which are snoc-lists of types. We also have a `conc` operation that concatenates two contexts. The `term` datatype represents in fact the possible typing derivations of the calculus, which are isomorphic to the well-typed terms, hence the name. A term is either an application of:

- the axiom rule to type a reference to the first variable in a context,
- the weakening rule to type an object in a larger context

- the abstraction or lambda rule to type a function
- the application to type an application of a function to an argument

Once we have this datatype we want to do proofs on it, like weakening:

```
Coq < Lemma weakening : forall G D tau, term (G ; D) tau ->
Coq <   forall tau', term (G , tau' ; D) tau.
```

The problem here is that we can't just use `induction` on the typing derivation because it will forget about the `G ; D` constraint appearing in the instance. A solution would be to rewrite the goal as:

```
Coq < Lemma weakening' : forall G' tau, term G' tau ->
Coq <   forall G D, (G ; D) = G' ->
Coq <   forall tau', term (G, tau' ; D) tau.
```

With this proper separation of the index from the instance and the right induction loading (putting `G` and `D` after the inducted-on hypothesis), the proof will go through, but it is a very tedious process. One is also forced to make a wrapper lemma to get back the more natural statement. The `dependent induction` tactic aleviates this trouble by doing all of this plumbing of generalizing and substituting back automatically. Indeed we can simply write:

```
Coq < Require Import Coq.Program.Tactics.
```

```
Coq < Lemma weakening : forall G D tau, term (G ; D) tau ->
Coq <   forall tau', term (G , tau' ; D) tau.
```

```
Coq < Proof with simpl in * ; simpl_depind ; auto.
```

```
Coq <   intros G D tau H. dependent induction H generalizing G D ; intros.
```

This call to `dependent induction` has an additional arguments which is a list of variables appearing in the instance that should be generalized in the goal, so that they can vary in the induction hypotheses. By default, all variables appearing inside constructors (except in a parameter position) of the instantiated hypothesis will be generalized automatically but one can always give the list explicitely.

```
Coq <   Show.
4 subgoals

  G : ctx
  tau : type
  G0 : ctx
  D : ctx
  H : G, tau = G0; D
  tau' : type
  ============================
   term ((G0, tau'); D) tau
subgoal 2 is:
 term ((G0, tau'0); D) tau
subgoal 3 is:
 term ((G0, tau'0); D) (tau -> tau')
subgoal 4 is:
 term ((G0, tau'0); D) tau'
```

The simpl_depind tactic includes an automatic tactic that tries to simplify equalities appearing at the beginning of induction hypotheses, generally using trivial applications of reflexiviy. In cases where the equality is not between constructor forms though, one must help the automation by giving some arguments, using the specialize tactic.

```
Coq < destruct D... apply weak ; apply ax. apply ax.

Coq < destruct D...

Coq < Show.
4 subgoals

  G : ctx
  tau : type
  H : term G tau
  tau' : type
  IHterm : forall G0 D : ctx,
           G = G0; D -> forall tau' : type, term ((G0, tau'); D) tau
  tau'0 : type
  ============================
   term ((G, tau'), tau'0) tau
subgoal 2 is:
 term (((G0, tau'0); D), t) tau
subgoal 3 is:
 term ((G0, tau'0); D) (tau -> tau')
subgoal 4 is:
 term ((G0, tau'0); D) tau'

Coq <   specialize (IHterm G empty).
4 subgoals

  G : ctx
  tau : type
  H : term G tau
  tau' : type
  IHterm : G = G; empty -> forall tau' : type, term ((G, tau'); empty) tau
  tau'0 : type
  ============================
   term ((G, tau'), tau'0) tau
subgoal 2 is:
 term (((G0, tau'0); D), t) tau
subgoal 3 is:
 term ((G0, tau'0); D) (tau -> tau')
subgoal 4 is:
 term ((G0, tau'0); D) tau'
```

Then the automation can find the needed equality G = G to narrow the induction hypothesis further. This concludes our example.

```
Coq <   simpl_depind.
4 subgoals

  G : ctx
  tau : type
```

```
  H : term G tau
  tau' : type
  tau'0 : type
  IHterm : forall tau' : type, term ((G, tau'); empty) tau
  ============================
   term ((G, tau'), tau'0) tau
subgoal 2 is:
 term (((G0, tau'0); D), t) tau
subgoal 3 is:
 term ((G0, tau'0); D) (tau -> tau')
subgoal 4 is:
 term ((G0, tau'0); D) tau'
```

See also: The induction 11, case 9 and inversion 8.10 tactics.

10.7 autorewrite

Here are two examples of `autorewrite` use. The first one (*Ackermann function*) shows actually a quite basic use where there is no conditional rewriting. The second one (*Mac Carthy function*) involves conditional rewritings and shows how to deal with them using the optional tactic of the `Hint Rewrite` command.

Example 1: *Ackermann function*

```
Coq < Require Import Arith.

Coq < Variable Ack :
Coq <            nat -> nat -> nat.

Coq < Axiom Ack0 :
Coq <         forall m:nat, Ack 0 m = S m.

Coq < Axiom Ack1 : forall n:nat, Ack (S n) 0 = Ack n 1.

Coq < Axiom Ack2 : forall n m:nat, Ack (S n) (S m) = Ack n (Ack (S n) m).


Coq < Hint Rewrite Ack0 Ack1 Ack2 : base0.
Toplevel input, characters 13-17:
> Hint Rewrite Ack0 Ack1 Ack2 : base0.
>              ^^^^
Error: The reference Ack0 was not found in the current environment.

Coq < Lemma ResAck0 :
Coq <  Ack 3 2 = 29.
Toplevel input, characters 18-21:
>  Ack 3 2 = 29.
>  ^^^
Error: The reference Ack was not found in the current environment.

Coq < autorewrite with base0 using try reflexivity.
Toplevel input, characters 0-44:
> autorewrite with base0 using try reflexivity.
> ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: Rewriting base base0 does not exist.
```

Example 2: *Mac Carthy function*

```
Coq < Require Import Omega.

Coq < Variable g :
Coq <            nat -> nat -> nat.

Coq < Axiom g0 :
Coq <          forall m:nat, g 0 m = m.

Coq < Axiom
Coq <   g1 :
Coq <     forall n m:nat,
Coq <       (n > 0) -> (m > 100) -> g n m = g (pred n) (m - 10).

Coq < Axiom
Coq <   g2 :
Coq <     forall n m:nat,
Coq <       (n > 0) -> (m <= 100) -> g n m = g (S n) (m + 11).

Coq < Hint Rewrite g0 g1 g2 using omega : base1.

Coq < Lemma Resg0 :
Coq <  g 1 110 = 100.
1 subgoal

  ============================
   g 1 110 = 100

Coq < autorewrite with base1 using reflexivity || simpl.
Proof completed.

Coq < Lemma Resg1 : g 1 95 = 91.
1 subgoal

  ============================
   g 1 95 = 91

Coq < autorewrite with base1 using reflexivity || simpl.
Proof completed.
```

10.8 quote

The tactic `quote` allows to use Barendregt's so-called 2-level approach without writing any ML code. Suppose you have a language `L` of 'abstract terms' and a type `A` of 'concrete terms' and a function `f : L -> A`. If `L` is a simple inductive datatype and `f` a simple fixpoint, `quote f` will replace the head of current goal by a convertible term of the form `(f t)`. `L` must have a constructor of type: `A -> L`.

Here is an example:

```
Coq < Require Import Quote.

Coq < Parameters A B C : Prop.
A is assumed
B is assumed
C is assumed

Coq < Inductive formula : Type :=
```

```
Coq <   | f_and : formula -> formula -> formula (* binary constructor *)
Coq <   | f_or : formula -> formula -> formula
Coq <   | f_not : formula -> formula (* unary constructor *)
Coq <   | f_true : formula (* 0-ary constructor *)
Coq <   | f_const : Prop -> formula (* contructor for constants *).
formula is defined
formula_rect is defined
formula_ind is defined
formula_rec is defined
Coq < Fixpoint interp_f (f:
Coq <                    formula) : Prop :=
Coq <   match f with
Coq <   | f_and f1 f2 => interp_f f1 /\ interp_f f2
Coq <   | f_or f1 f2 => interp_f f1 \/ interp_f f2
Coq <   | f_not f1 => ~ interp_f f1
Coq <   | f_true => True
Coq <   | f_const c => c
Coq <   end.
interp_f is recursively defined (decreasing on 1st argument)
Coq < Goal A /\ (A \/ True) /\ ~ B /\ (A <-> A).
1 subgoal

  ============================
   A /\ (A \/ True) /\ ~ B /\ (A <-> A)
Coq < quote interp_f.
1 subgoal

  ============================
   interp_f
     (f_and (f_const A)
        (f_and (f_or (f_const A) f_true)
           (f_and (f_not (f_const B)) (f_const (A <-> A)))))
```

The algorithm to perform this inversion is: try to match the term with right-hand sides expression of `f`. If there is a match, apply the corresponding left-hand side and call yourself recursively on sub-terms. If there is no match, we are at a leaf: return the corresponding constructor (here `f_const`) applied to the term.

Error messages:

1. `quote: not a simple fixpoint`
   Happens when `quote` is not able to perform inversion properly.

10.8.1 Introducing variables map

The normal use of `quote` is to make proofs by reflection: one defines a function `simplify : formula -> formula` and proves a theorem `simplify_ok: (f:formula)(interp_f (simplify f)) -> (interp_f f)`. Then, one can simplify formulas by doing:

```
quote interp_f.
apply simplify_ok.
compute.
```

But there is a problem with leafs: in the example above one cannot write a function that implements, for example, the logical simplifications $A \wedge A \rightarrow A$ or $A \wedge \neg A \rightarrow$ `False`. This is because the Prop is impredicative.

It is better to use that type of formulas:

```
Coq < Inductive formula : Set :=
Coq <   | f_and : formula -> formula -> formula
Coq <   | f_or : formula -> formula -> formula
Coq <   | f_not : formula -> formula
Coq <   | f_true : formula
Coq <   | f_atom : index -> formula.
formula is defined
formula_rect is defined
formula_ind is defined
formula_rec is defined
```

`index` is defined in module `quote`. Equality on that type is decidable so we are able to simplify $A \wedge A$ into A at the abstract level.

When there are variables, there are bindings, and `quote` provides also a type `(varmap A)` of bindings from `index` to any set `A`, and a function `varmap_find` to search in such maps. The interpretation function has now another argument, a variables map:

```
Coq < Fixpoint interp_f (vm:
Coq <                      varmap Prop) (f:formula) {struct f} : Prop :=
Coq <   match f with
Coq <   | f_and f1 f2 => interp_f vm f1 /\ interp_f vm f2
Coq <   | f_or f1 f2 => interp_f vm f1 \/ interp_f vm f2
Coq <   | f_not f1 => ~ interp_f vm f1
Coq <   | f_true => True
Coq <   | f_atom i => varmap_find True i vm
Coq <   end.
interp_f is recursively defined (decreasing on 2nd argument)
```

`quote` handles this second case properly:

```
Coq < Goal A /\ (B \/ A) /\ (A \/ ~ B).
1 subgoal

  ============================
   A /\ (B \/ A) /\ (A \/ ~ B)

Coq < quote interp_f.
1 subgoal

  ============================
   interp_f
     (Node_vm B (Node_vm A (Empty_vm Prop) (Empty_vm Prop)) (Empty_vm Prop))
     (f_and (f_atom (Left_idx End_idx))
        (f_and (f_or (f_atom End_idx) (f_atom (Left_idx End_idx)))
           (f_or (f_atom (Left_idx End_idx)) (f_not (f_atom End_idx)))))
```

It builds `vm` and `t` such that `(f vm t)` is convertible with the conclusion of current goal.

10.8.2 Combining variables and constants

One can have both variables and constants in abstracts terms; that is the case, for example, for the `ring` tactic (chapter 23). Then one must provide to `quote` a list of *constructors of constants*. For example, if the list is `[O S]` then closed natural numbers will be considered as constants and other terms as variables.

Example:

```
Coq < Inductive formula : Type :=
Coq <   | f_and : formula -> formula -> formula
Coq <   | f_or : formula -> formula -> formula
Coq <   | f_not : formula -> formula
Coq <   | f_true : formula
Coq <   | f_const : Prop -> formula (* constructor for constants *)
Coq <   | f_atom : index -> formula.

Coq < Fixpoint interp_f
Coq <  (vm:            (* constructor for variables *)
Coq <   varmap Prop) (f:formula) {struct f} : Prop :=
Coq <   match f with
Coq <   | f_and f1 f2 => interp_f vm f1 /\ interp_f vm f2
Coq <   | f_or f1 f2 => interp_f vm f1 \/ interp_f vm f2
Coq <   | f_not f1 => ~ interp_f vm f1
Coq <   | f_true => True
Coq <   | f_const c => c
Coq <   | f_atom i => varmap_find True i vm
Coq <   end.

Coq < Goal
Coq < A /\ (A \/ True) /\ ~ B /\ (C <-> C).

Coq < quote interp_f [ A B ].
1 subgoal

  ============================
   interp_f (Node_vm (C <-> C) (Empty_vm Prop) (Empty_vm Prop))
     (f_and (f_const A)
        (f_and (f_or (f_const A) f_true)
           (f_and (f_not (f_const B)) (f_atom End_idx))))

Coq < Undo.
1 subgoal

  ============================
   A /\ (A \/ True) /\ ~ B /\ (C <-> C)

Coq <   quote interp_f [ B C iff ].
1 subgoal

  ============================
   interp_f (Node_vm A (Empty_vm Prop) (Empty_vm Prop))
     (f_and (f_atom End_idx)
        (f_and (f_or (f_atom End_idx) f_true)
           (f_and (f_not (f_const B)) (f_const (C <-> C)))))
```

```
Coq < Section Sort.

Coq < Variable A : Set.

Coq < Inductive permut : list A -> list A -> Prop :=
Coq <   | permut_refl   : forall l, permut l l
Coq <   | permut_cons   :
Coq <       forall a l0 l1, permut l0 l1 -> permut (a :: l0) (a :: l1)
Coq <   | permut_append : forall a l, permut (a :: l) (l ++ a :: nil)
Coq <   | permut_trans  :
Coq <       forall l0 l1 l2, permut l0 l1 -> permut l1 l2 -> permut l0 l2.

Coq < End Sort.
```

Figure 10.1: Definition of the permutation predicate

Warning: Since function inversion is undecidable in general case, don't expect miracles from it!

See also: comments of source file `tactics/contrib/polynom/quote.ml`

See also: the `ring` tactic (Chapter 23)

10.9 Using the tactical language

10.9.1 About the cardinality of the set of natural numbers

A first example which shows how to use the pattern matching over the proof contexts is the proof that natural numbers have more than two elements. The proof of such a lemma can be done as follows:

```
Coq < Lemma card_nat :
Coq <  ~ (exists x : nat, exists y : nat, forall z:nat, x = z \/ y = z).

Coq < Proof.

Coq < red; intros (x, (y, Hy)).

Coq < elim (Hy 0); elim (Hy 1); elim (Hy 2); intros;
Coq <  match goal with
Coq <  | [_:(?a = ?b),_:(?a = ?c) |- _ ] =>
Coq <      cut (b = c); [ discriminate | apply trans_equal with a; auto ]
Coq <  end.

Coq < Qed.
```

We can notice that all the (very similar) cases coming from the three eliminations (with three distinct natural numbers) are successfully solved by a `match goal` structure and, in particular, with only one pattern (use of non-linear matching).

10.9.2 Permutation on closed lists

Another more complex example is the problem of permutation on closed lists. The aim is to show that a closed list is a permutation of another one.

First, we define the permutation predicate as shown in table 10.1.

A more complex example is the problem of permutation on closed lists. The aim is to show that a closed list is a permutation of another one. First, we define the permutation predicate as shown on Figure 10.1.

```
Coq < Ltac Permut n :=
Coq <   match goal with
Coq <   | |- (permut _ ?l ?l) => apply permut_refl
Coq <   | |- (permut _ (?a :: ?l1) (?a :: ?l2)) =>
Coq <       let newn := eval compute in (length l1) in
Coq <       (apply permut_cons; Permut newn)
Coq <   | |- (permut ?A (?a :: ?l1) ?l2) =>
Coq <       match eval compute in n with
Coq <       | 1 => fail
Coq <       | _ =>
Coq <           let l1' := constr:(l1 ++ a :: nil) in
Coq <           (apply (permut_trans A (a :: l1) l1' l2);
Coq <             [ apply permut_append | compute; Permut (pred n) ])
Coq <       end
Coq <   end.
Permut is defined

Coq < Ltac PermutProve :=
Coq <   match goal with
Coq <   | |- (permut _ ?l1 ?l2) =>
Coq <       match eval compute in (length l1 = length l2) with
Coq <       | (?n = ?n) => Permut n
Coq <       end
Coq <   end.
PermutProve is defined
```

Figure 10.2: Permutation tactic

Next, we can write naturally the tactic and the result can be seen on Figure 10.2. We can notice that we use two toplevel definitions `PermutProve` and `Permut`. The function to be called is `PermutProve` which computes the lengths of the two lists and calls `Permut` with the length if the two lists have the same length. `Permut` works as expected. If the two lists are equal, it concludes. Otherwise, if the lists have identical first elements, it applies `Permut` on the tail of the lists. Finally, if the lists have different first elements, it puts the first element of one of the lists (here the second one which appears in the `permut` predicate) at the end if that is possible, i.e., if the new first element has been at this place previously. To verify that all rotations have been done for a list, we use the length of the list as an argument for `Permut` and this length is decremented for each rotation down to, but not including, 1 because for a list of length n, we can make exactly $n-1$ rotations to generate at most n distinct lists. Here, it must be noticed that we use the natural numbers of Coq for the rotation counter. On Figure 9.1, we can see that it is possible to use usual natural numbers but they are only used as arguments for primitive tactics and they cannot be handled, in particular, we cannot make computations with them. So, a natural choice is to use Coq data structures so that Coq makes the computations (reductions) by `eval compute in` and we can get the terms back by `match`.

With `PermutProve`, we can now prove lemmas as follows:

```
Coq < Lemma permut_ex1 :
Coq <   permut nat (1 :: 2 :: 3 :: nil) (3 :: 2 :: 1 :: nil).

Coq < Proof. PermutProve. Qed.

Coq < Lemma permut_ex2 :
Coq <   permut nat
Coq <     (0 :: 1 :: 2 :: 3 :: 4 :: 5 :: 6 :: 7 :: 8 :: 9 :: nil)
```

```
Coq <      (0 :: 2 :: 4 :: 6 :: 8 :: 9 :: 7 :: 5 :: 3 :: 1 :: nil).
Coq < Proof. PermutProve. Qed.
```

10.9.3 Deciding intuitionistic propositional logic

The pattern matching on goals allows a complete and so a powerful backtracking when returning tactic values. An interesting application is the problem of deciding intuitionistic propositional logic. Considering the contraction-free sequent calculi `LJT*` of Roy Dyckhoff ([54]), it is quite natural to code such a tactic using the tactic language as shown on Figures 10.3 and 10.4. The tactic `Axioms` tries to conclude using usual axioms. The tactic `DSimplif` applies all the reversible rules of Dyckhoff's system. Finally, the tactic `TautoProp` (the main tactic to be called) simplifies with `DSimplif`, tries to conclude with `Axioms` and tries several paths using the backtracking rules (one of the four Dyckhoff's rules for the left implication to get rid of the contraction and the right or).

For example, with `TautoProp`, we can prove tautologies like those:

```
Coq < Lemma tauto_ex1 : forall A B:Prop, A /\ B -> A \/ B.
Coq < Proof. TautoProp. Qed.
Coq < Lemma tauto_ex2 :
Coq <    forall A B:Prop, (~ ~ B -> B) -> (A -> B) -> ~ ~ A -> B.
Coq < Proof. TautoProp. Qed.
```

10.9.4 Deciding type isomorphisms

A more tricky problem is to decide equalities between types and modulo isomorphisms. Here, we choose to use the isomorphisms of the simply typed λ-calculus with Cartesian product and $unit$ type (see, for example, [43]). The axioms of this λ-calculus are given by table 10.5.

A more tricky problem is to decide equalities between types and modulo isomorphisms. Here, we choose to use the isomorphisms of the simply typed λ-calculus with Cartesian product and $unit$ type (see, for example, [43]). The axioms of this λ-calculus are given on Figure 10.5.

The tactic to judge equalities modulo this axiomatization can be written as shown on Figures 10.6 and 10.7. The algorithm is quite simple. Types are reduced using axioms that can be oriented (this done by `MainSimplif`). The normal forms are sequences of Cartesian products without Cartesian product in the left component. These normal forms are then compared modulo permutation of the components (this is done by `CompareStruct`). The main tactic to be called and realizing this algorithm is `IsoProve`.

Here are examples of what can be solved by `IsoProve`.

```
Coq < Lemma isos_ex1 :
```

```
Coq < Ltac Axioms :=
Coq <   match goal with
Coq <   | |- True => trivial
Coq <   | _:False |- _  => elimtype False; assumption
Coq <   | _:?A |- ?A  => auto
Coq <   end.
Axioms is defined
```

Figure 10.3: Deciding intuitionistic propositions (1)

```
Coq < Ltac DSimplif :=
Coq <   repeat
Coq <    (intros;
Coq <     match goal with
Coq <       | id:(~ _) |- _ => red in id
Coq <       | id:(_ /\ _) |- _ =>
Coq <           elim id; do 2 intro; clear id
Coq <       | id:(_ \/ _) |- _ =>
Coq <           elim id; intro; clear id
Coq <       | id:(?A /\ ?B -> ?C) |- _ =>
Coq <           cut (A -> B -> C);
Coq <            [ intro | intros; apply id; split; assumption ]
Coq <       | id:(?A \/ ?B -> ?C) |- _ =>
Coq <           cut (B -> C);
Coq <            [ cut (A -> C);
Coq <               [ intros; clear id
Coq <               | intro; apply id; left; assumption ]
Coq <            | intro; apply id; right; assumption ]
Coq <       | id0:(?A -> ?B),id1:?A |- _ =>
Coq <           cut B; [ intro; clear id0 | apply id0; assumption ]
Coq <       | |- (_ /\ _) => split
Coq <       | |- (~ _) => red
Coq <      end).
DSimplif is defined
Coq < Ltac TautoProp :=
Coq <   DSimplif;
Coq <    Axioms ||
Coq <      match goal with
Coq <      | id:((?A -> ?B) -> ?C) |- _ =>
Coq <           cut (B -> C);
Coq <           [ intro; cut (A -> B);
Coq <              [ intro; cut C;
Coq <                 [ intro; clear id | apply id; assumption ]
Coq <              | clear id ]
Coq <           | intro; apply id; intro; assumption ]; TautoProp
Coq <      | id:(~ ?A -> ?B) |- _ =>
Coq <          cut (False -> B);
Coq <           [ intro; cut (A -> False);
Coq <              [ intro; cut B;
Coq <                 [ intro; clear id | apply id; assumption ]
Coq <              | clear id ]
Coq <           | intro; apply id; red; intro; assumption ]; TautoProp
Coq <      | |- (_ \/ _) => (left; TautoProp) || (right; TautoProp)
Coq <      end.
TautoProp is defined
```

Figure 10.4: Deciding intuitionistic propositions (2)

```
Coq <   forall A B:Set, A * unit * B = B * (unit * A).

Coq < Proof.

Coq < intros; IsoProve.
```

```
Coq < Open Scope type_scope.
Coq < Section Iso_axioms.
Coq < Variables A B C : Set.
Coq < Axiom Com : A * B = B * A.
Coq < Axiom Ass : A * (B * C) = A * B * C.
Coq < Axiom Cur : (A * B -> C) = (A -> B -> C).
Coq < Axiom Dis : (A -> B * C) = (A -> B) * (A -> C).
Coq < Axiom P_unit : A * unit = A.
Coq < Axiom AR_unit : (A -> unit) = unit.
Coq < Axiom AL_unit : (unit -> A) = A.
Coq < Lemma Cons : B = C -> A * B = A * C.
Coq < Proof.
Coq < intro Heq; rewrite Heq; apply refl_equal.
Coq < Qed.
Coq < End Iso_axioms.
```

Figure 10.5: Type isomorphism axioms

```
Coq < Qed.
Coq <
Coq < Lemma isos_ex2 :
Coq <   forall A B C:Set,
Coq <     (A * unit -> B * (C * unit)) =
Coq <     (A * unit -> (C -> unit) * C) * (unit -> A -> B).
Coq < Proof.
Coq < intros; IsoProve.
Coq < Qed.
```

```
Coq < Ltac DSimplif trm :=
Coq <   match trm with
Coq <    | (?A * ?B * ?C) =>
Coq <        rewrite <- (Ass A B C); try MainSimplif
Coq <    | (?A * ?B -> ?C) =>
Coq <        rewrite (Cur A B C); try MainSimplif
Coq <    | (?A -> ?B * ?C) =>
Coq <        rewrite (Dis A B C); try MainSimplif
Coq <    | (?A * unit) =>
Coq <        rewrite (P_unit A); try MainSimplif
Coq <    | (unit * ?B) =>
Coq <        rewrite (Com unit B); try MainSimplif
Coq <    | (?A -> unit) =>
Coq <        rewrite (AR_unit A); try MainSimplif
Coq <    | (unit -> ?B) =>
Coq <        rewrite (AL_unit B); try MainSimplif
Coq <    | (?A * ?B) =>
Coq <        (DSimplif A; try MainSimplif) || (DSimplif B; try MainSimplif)
Coq <    | (?A -> ?B) =>
Coq <        (DSimplif A; try MainSimplif) || (DSimplif B; try MainSimplif)
Coq <   end
Coq <  with MainSimplif :=
Coq <   match goal with
Coq <    | |- (?A = ?B) => try DSimplif A; try DSimplif B
Coq <   end.
DSimplif is defined
MainSimplif is defined

Coq < Ltac Length trm :=
Coq <   match trm with
Coq <    | (_ * ?B) => let succ := Length B in constr:(S succ)
Coq <    | _ => constr:1
Coq <   end.
Length is defined

Coq < Ltac assoc := repeat rewrite <- Ass.
assoc is defined
```

Figure 10.6: Type isomorphism tactic (1)

```
Coq < Ltac DoCompare n :=
Coq <   match goal with
Coq <   | [ |- (?A = ?A) ] => apply refl_equal
Coq <   | [ |- (?A * ?B = ?A * ?C) ] =>
Coq <       apply Cons; let newn := Length B in
Coq <                   DoCompare newn
Coq <   | [ |- (?A * ?B = ?C) ] =>
Coq <       match eval compute in n with
Coq <       | 1 => fail
Coq <       | _ =>
Coq <           pattern (A * B) at 1; rewrite Com; assoc; DoCompare (pred n)
Coq <       end
Coq <   end.
DoCompare is defined

Coq < Ltac CompareStruct :=
Coq <   match goal with
Coq <   | [ |- (?A = ?B) ] =>
Coq <       let l1 := Length A
Coq <       with l2 := Length B in
Coq <       match eval compute in (l1 = l2) with
Coq <       | (?n = ?n) => DoCompare n
Coq <       end
Coq <   end.
CompareStruct is defined

Coq < Ltac IsoProve := MainSimplif; CompareStruct.
IsoProve is defined
```

Figure 10.7: Type isomorphism tactic (2)

Chapter 11

The C-zar mathematical proof language

11.1 Introduction

11.1.1 Foreword

In this chapter, we describe an alternative language that may be used to do proofs using the Coq proof assistant. The language described here uses the same objects (proof-terms) as Coq, but it differs in the way proofs are described. This language was created by Pierre Corbineau at the Radboud University of Nijmegen, The Netherlands.

The intent is to provide a language where proofs are less formalism- and implementation-sensitive, and in the process to ease a bit the learning of computer-aided proof verification.

11.1.2 What is a declarative proof ?

In vanilla Coq, proofs are written in the imperative style: the user issues commands that transform a so called proof state until it reaches a state where the proof is completed. In the process, the user mostly described the transitions of this system rather than the intermediate states it goes through.

The purpose of a declarative proof language is to take the opposite approach where intermediate states are always given by the user, but the transitions of the system are automated as much as possible.

While not being a purely declarative language, the C-zar mathematical proof language aims at providing a solution for users who wish to edit Coq proofs following the declarative philosophy.

11.1.3 Well-formedness and Completeness

The C-zar mathematical proof language introduces a notion of well-formed proofs which are weaker than correct (and complete) proofs. Well-formed proofs are actually proof script where only the reasoning is incomplete. All the other aspects of the proof are correct:

- All objects referred to exist where they are used
- Conclusion steps actually prove something related to the conclusion of the theorem (the `thesis`.
- Hypothesis introduction steps are done when the goal is an implication with a corresponding assumption.
- Sub-objects in the elimination steps for tuples are correct sub-objects of the tuple being decomposed.

- Patterns in case analysis are type-correct, and induction is well guarded.

11.1.4 Note for tactics users

This section explain what differences the casual Coq user will experience using the C-zar mathematical proof language.

1. The focusing mechanism is constrained so that only one goal at a time is visible.

2. Giving a statement that Coq cannot prove does not produce an error, only a warning: this allows to go on with the proof and fill the gap later.

3. Tactics can still be used for justifications and after `escape`.

11.1.5 Compatibility

The C-zar mathematical proof language is available for all Coq interfaces that use text-based interaction, including:

- the command-line toplevel `coqtop`
- the native GUI `coqide`
- the Proof-General emacs mode
- Cezary Kaliszyk's Web interface
- L.E. Mamane's tmEgg TeXmacs plugin

However it is not supported by structured editors such as PCoq.

11.2 Syntax

Here is a complete formal description of the syntax for C-zar commands.

The lexical conventions used here follows those of section 1.1.

Conventions:

- `<tactic>` stands for an Coq tactic.

11.2.1 Temporary names

In proof commands where an optional name is asked for, omitting the name will trigger the creation of a fresh temporary name (e.g. for a hypothesis). Temporary names always start with an underscore '_' character (e.g. `_hyp0`). Temporary names have a lifespan of one command: they get erased after the next command. They can however be used safely in the step after their creation.

instruction	::=	`proof`
	|	`assume` statement `and` ... `and` statement *[*[`and` {`we have`}-clause]*]*
	|	{`let`,`be`}-clause
	|	{`given`}-clause
	|	{`consider`}-clause `from` term
	|	(`have` | `then` | `thus` | `hence`]) statement justification
	|	*[*`thus`*]* (∼= | =∼) *[ident* **:** *]term* `justification`
	|	`suffices` ({`to have`}-clause | statement `and` ... `and` statement *[*`and` {`to have`}-clau
		`to show` statement justification
	|	(`claim` | `focus on`) statement
	|	`take` *term*
	|	`define` *ident[*var **,** ... **,** var*]* `as` *term*
	|	`reconsider` (*ident*| `thesis`) `as` type
	|	`per` (`cases`|`induction`) `on` *term*
	|	`per cases of` type justification
	|	`suppose` *[*ident **,** ... **,** ident `and`*]* `it is` pattern
		[`such that` statement `and` ... `and` statement *[*`and` {`we have`}-clause*]]*
	|	`end` (`proof` | `claim` | `focus` | `cases` | `induction`)
	|	`escape`
	|	`return`
$\{\alpha,\beta\}$-clause	::=	α var **,** ... **,** var β `such that` statement `and` ... `and` statement
		[`and` $\{\alpha,\beta\}$-clause*]*
statement	::=	*[ident* **:** *]* type
	|	`thesis`
	|	`thesis for` *ident*
var	::=	*ident[***:** type*]*
justification	::=	*[*`by` (⋆ | *term* **,** ... **,** *term*)*]* *[*`using` tactic*]*

Figure 11.1: Syntax of mathematical proof commands

11.3 Language description

11.3.1 Starting and Ending a mathematical proof

The standard way to use the C-zar mathematical proof language is to first state a `Lemma/Theorem/Definition` and then use the `proof` command to switch the current subgoal to mathematical mode. After the proof is completed, the `end proof` command will close the mathematical proof. If any subgoal remains to be proved, they will be displayed using the usual Coq display.

```
Coq < Theorem this_is_trivial: True.
1 subgoal

  ============================
   True

Coq < proof.
```

```
1 subgoal
     *** Declarative Mode ***

  ============================
  thesis :=
   True
Coq <   thus thesis.
Subproof completed, now type "end proof".
Coq < end proof.
Proof completed.
Coq < Qed.
proof.
     thus thesis.
end proof.
this_is_trivial is defined
```

The `proof` command only applies to *one subgoal*, thus if several sub-goals are already present, the `proof .. end proof` sequence has to be used several times.

```
Coq <   Show.
3 subgoals

  ============================
   True
subgoal 2 is:
 True
subgoal 3 is:
 True
Coq <   proof. (* first subgoal *)
1 subgoal
     *** Declarative Mode ***

  ============================
  thesis :=
   True
Coq <     thus thesis.
Subproof completed, now type "end proof".
Coq <   end proof.
2 subgoals

  ============================
   True
subgoal 2 is:
 True
Coq <   trivial. (* second subgoal *)
1 subgoal

  ============================
   True
Coq <   proof. (* third subgoal *)
```

```
1 subgoal
     *** Declarative Mode ***

  ============================
  thesis :=
   True

Coq <     thus thesis.
Subproof completed, now type "end proof".

Coq <   end proof.
Proof completed.
```

As with all other block structures, the `end proof` command assumes that your proof is complete. If not, executing it will be equivalent to admitting that the statement is proved: A warning will be issued and you will not be able to run the `Qed` command. Instead, you can run `Admitted` if you wish to start another theorem and come back later.

```
Coq < Theorem this_is_not_so_trivial: False.
1 subgoal

  ============================
   False

Coq < proof.
1 subgoal
     *** Declarative Mode ***

  ============================
  thesis :=
   False

Coq < end proof. (* here a warning is issued *)
Proof completed.

Coq < Qed. (* fails : the proof in incomplete *)
proof.
(* Some proof has been skipped here *)
end proof.
Error: Attempt to save an incomplete proof

Coq < Admitted. (* Oops! *)
this_is_not_so_trivial is assumed
```

11.3.2 Switching modes

When writing a mathematical proof, you may wish to use procedural tactics at some point. One way to do so is to write a using-phrase in a deduction step (see section 11.3.14). The other way is to use an `escape...return` block.

```
Coq <  Show.
1 subgoal
     *** Declarative Mode ***

  ============================
```

```
  thesis :=
   True
Coq <  escape.
1 subgoal

  ============================
   True
Coq <  auto.
Subgoal proved
Subproof completed, now type "return".
Coq <  return.
Subproof completed, now type "end proof".
```

The return statement expects all subgoals to be closed, otherwise a warning is issued and the proof cannot be saved anymore.

It is possible to use the `proof` command inside an `escape...return` block, thus nesting a mathematical proof inside a procedural proof inside a mathematical proof ...

11.3.3 Computation steps

The `reconsider ... as` command allows to change the type of a hypothesis or of `thesis` to a convertible one.

```
Coq <  Show.
1 subgoal
     *** Declarative Mode ***

  a := false : bool
  b := true : bool
  H : if a then True else False
  ============================
  thesis :=
   if b then True else False
Coq <  reconsider H as False.
1 subgoal
     *** Declarative Mode ***

  a := false : bool
  b := true : bool
  H : False
  ============================
  thesis :=
   if b then True else False
Coq <  reconsider thesis as True.
1 subgoal
     *** Declarative Mode ***

  a := false : bool
  b := true : bool
  H : False
```

```
  ============================
  thesis :=
   True
```

11.3.4 Deduction steps

The most common instruction in a mathematical proof is the deduction step: it asserts a new statement (a formula/type of the pCIC) and tries to prove it using a user-provided indication : the justification. The asserted statement is then added as a hypothesis to the proof context.

```
Coq <  Show.
1 subgoal
     *** Declarative Mode ***

  x : nat
  H : x = 2
  ============================
  thesis :=
   2 + x = 4
Coq <  have H':(2+x=2+2) by H.
1 subgoal
     *** Declarative Mode ***

  x : nat
  H : x = 2
  H' : 2 + x = 2 + 2
  ============================
  thesis :=
   2 + x = 4
```

It is very often the case that the justifications uses the last hypothesis introduced in the context, so the `then` keyword can be used as a shortcut, e.g. if we want to do the same as the last example :

```
Coq <  Show.
1 subgoal
     *** Declarative Mode ***

  x : nat
  H : x = 2
  ============================
  thesis :=
   2 + x = 4
Coq <  then (2+x=2+2).
1 subgoal
     *** Declarative Mode ***

  x : nat
  H : x = 2
  _fact : 2 + x = 2 + 2
  ============================
  thesis :=
   2 + x = 4
```

In this example, you can also see the creation of a temporary name `_fact`.

11.3.5 Iterated equalities

A common proof pattern when doing a chain of deductions, is to do multiple rewriting steps over the same term, thus proving the corresponding equalities. The iterated equalities are a syntactic support for this kind of reasoning:

```
Coq <  Show.
1 subgoal
     *** Declarative Mode ***

  x : nat
  H : x = 2
  ============================
  thesis :=
   x + x = x * x

Coq <  have (4 = 4).
1 subgoal
     *** Declarative Mode ***

  x : nat
  H : x = 2
  _fact : 4 = 4
  ============================
  thesis :=
   x + x = x * x

Coq <         ~= (2 * 2).
1 subgoal
     *** Declarative Mode ***

  x : nat
  H : x = 2
  _eq : 4 = 2 * 2
  ============================
  thesis :=
   x + x = x * x

Coq <         ~= (x * x) by H.
1 subgoal
     *** Declarative Mode ***

  x : nat
  H : x = 2
  _eq0 : 4 = x * x
  ============================
  thesis :=
   x + x = x * x

Coq <         =~ (2 + 2).
1 subgoal
     *** Declarative Mode ***

  x : nat
  H : x = 2
```

```
  _eq : 2 + 2 = x * x
  ============================
  thesis :=
   x + x = x * x

Coq <          =~ H':(x + x) by H.
1 subgoal
     *** Declarative Mode ***

  x : nat
  H : x = 2
  H' : x + x = x * x
  ============================
  thesis :=
   x + x = x * x
```

Notice that here we use temporary names heavily.

11.3.6 Subproofs

When an intermediate step in a proof gets too complicated or involves a well contained set of intermediate deductions, it can be useful to insert its proof as a subproof of the current proof. this is done by using the `claim ... end claim` pair of commands.

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  x : nat
  H : x + x = x * x
  ============================
  thesis :=
   x = 0 \/ x = 2

Coq < claim H':((x - 2) * x = 0).
1 subgoal
     *** Declarative Mode ***

  x : nat
  H : x + x = x * x
  ============================
  thesis :=
   (x - 2) * x = 0
```

A few steps later ...

```
Coq < thus thesis.
Warning: Insufficient justification.
Subproof completed, now type "end claim".

Coq < end claim.
1 subgoal
     *** Declarative Mode ***
```

```
  x : nat
  H : x + x = x * x
  H' : (x - 2) * x = 0
  ============================
  thesis :=
   x = 0 \/ x = 2
```

Now the rest of the proof can happen.

11.3.7 Conclusion steps

The commands described above have a conclusion counterpart, where the new hypothesis is used to refine the conclusion.

Let us begin with simple examples :

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
  HA : A
  HB : B
  ============================
  thesis :=
   A /\ B
Coq < hence B.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
  HA : A
  HB : B
  _fact : B
  ============================
  thesis :=
   A
```

In the next example, we have to use `thus` because `HB` is no longer the last hypothesis.

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***
```

X	simple	with previous step	opens sub-proof	iterated equality
intermediate step	`have`	`then`	`claim`	`~=/=~`
conclusion step	`thus`	`hence`	`focus on`	`thus ~=/=~`

Figure 11.2: Correspondence between basic forward steps and conclusion steps

```
  A : Prop
  B : Prop
  C : Prop
  HA : A
  HB : B
  HC : C
  ============================
  thesis :=
   A /\ B /\ C
```

```
Coq < thus B by HB.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
  C : Prop
  HA : A
  HB : B
  HC : C
  _fact : B
  ============================
  thesis :=
   A /\ C
```

The command fails the refinement process cannot find a place to fit the object in a proof of the conclusion.

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
  C : Prop
  HA : A
  HB : B
  HC : C
  ============================
  thesis :=
   A /\ B
```

```
Coq < hence C. (* fails *)
Error: I could not relate this statement to the thesis.
```

The refinement process may induce non reversible choices, e.g. when proving a disjunction it may *choose* one side of the disjunction.

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
```

```
  HB : B
  ============================
  thesis :=
   A \/ B
```

```
Coq < hence B.
Subproof completed, now type "end proof".
```

In this example you can see that the right branch was chosen since `D` remains to be proved.

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
  C : Prop
  D : Prop
  HC : C
  HD : D
  ============================
  thesis :=
   A /\ B \/ C /\ D
```

```
Coq < thus C by HC.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
  C : Prop
  D : Prop
  HC : C
  HD : D
  _fact : C
  ============================
  thesis :=
   D
```

Now for existential statements, we can use the `take` command to choose `2` as an explicit witness of existence.

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  P : nat -> Prop
  HP : P 2
  ============================
  thesis :=
   exists x : nat, P x
```

```
Coq < take 2.
1 subgoal
     *** Declarative Mode ***
```

```
  P : nat -> Prop
  HP : P 2
  ============================
  thesis :=
   P 2
```

It is also possible to prove the existence directly.

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  P : nat -> Prop
  HP : P 2
  ============================
  thesis :=
   exists x : nat, P x

Coq < hence (P 2).
Subproof completed, now type "end proof".
```

Here a more involved example where the choice of `P 2` propagates the choice of `2` to another part of the formula.

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  P : nat -> Prop
  R : nat -> nat -> Prop
  HP : P 2
  HR : R 0 2
  ============================
  thesis :=
   exists x : nat, exists y : nat, P y /\ R x y

Coq < thus (P 2) by HP.
1 subgoal
     *** Declarative Mode ***

  P : nat -> Prop
  R : nat -> nat -> Prop
  HP : P 2
  HR : R 0 2
  _fact : P 2
  ============================
  thesis :=
   exists n : nat, R n 2
```

Now, an example with the `suffices` command. `suffices` is a sort of dual for `have`: it allows to replace the conclusion (or part of it) by a sufficient condition.

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
  P : nat -> Prop
  HP : forall x : nat, P x -> B
  HA : A
  ============================
  thesis :=
   A /\ B
Coq < suffices to have x such that HP':(P x) to show B by HP,HP'.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
  P : nat -> Prop
  HP : forall x : nat, P x -> B
  HA : A
  _cofact : forall x : nat, P x -> B
  ============================
  thesis :=
   A /\ (exists n : nat, P n)
```

Finally, an example where `focus` is handy : local assumptions.

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  P : nat -> Prop
  HP : P 2
  HA : A
  ============================
  thesis :=
   A /\ (forall x : nat, x = 2 -> P x)
Coq < focus on (forall x, x = 2 -> P x).
1 subgoal
     *** Declarative Mode ***

  A : Prop
  P : nat -> Prop
  HP : P 2
  HA : A
  ============================
  thesis :=
   forall x : nat, x = 2 -> P x
Coq < let x be such that (x = 2).
1 subgoal
```

```
     *** Declarative Mode ***

  A : Prop
  P : nat -> Prop
  HP : P 2
  HA : A
  x : nat
  _hyp : x = 2
  ============================
  thesis :=
   P x

Coq < hence thesis by HP.
Subproof completed, now type "end focus".

Coq < end focus.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  P : nat -> Prop
  HP : P 2
  HA : A
  _claim : forall x : nat, x = 2 -> P x
  ============================
  thesis :=
   A
```

11.3.8 Declaring an Abbreviation

In order to shorten long expressions, it is possible to use the `define ... as ...` command to give a name to recurring expressions.

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  x : nat
  H : x = 0
  ============================
  thesis :=
   x + x = x * x

Coq < define sqr x as (x * x).
1 subgoal
     *** Declarative Mode ***

  x : nat
  H : x = 0
  sqr := fun x : nat => x * x : nat -> nat
  ============================
  thesis :=
   x + x = x * x

Coq < reconsider thesis as (x + x = sqr x).
```

```
1 subgoal
     *** Declarative Mode ***

  x : nat
  H : x = 0
  sqr := fun x : nat => x * x : nat -> nat
  ============================
  thesis :=
   x + x = sqr x
```

11.3.9 Introduction steps

When the `thesis` consists of a hypothetical formula (implication or universal quantification (e.g. `A -> B`) , it is possible to assume the hypothetical part `A` and then prove `B`. In the C-zar mathematical proof language, this comes in two syntactic flavors that are semantically equivalent : `let` and `assume`. Their syntax is designed so that `let` works better for universal quantifiers and `assume` for implications.

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  P : nat -> Prop
  ============================
  thesis :=
   forall x : nat, P x -> P x
Coq < let x:nat.
1 subgoal
     *** Declarative Mode ***

  P : nat -> Prop
  x : nat
  ============================
  thesis :=
   P x -> P x
Coq < assume HP:(P x).
1 subgoal
     *** Declarative Mode ***

  P : nat -> Prop
  x : nat
  HP : P x
  ============================
  thesis :=
   P x
```

In the `let` variant, the type of the assumed object is optional provided it can be deduced from the command. The objects introduced by let can be followed by assumptions using `such that`.

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***
```

```
  P : nat -> Prop
  ============================
  thesis :=
   forall x : nat, P x -> P x
```

```
Coq < let x. (* fails because x's type is not clear *)
Toplevel input, characters 4-5:
> let x.
>     ^
Error: Cannot infer the type of x.
```

```
Coq < let x be such that HP:(P x). (* here x's type is inferred from (P x) *)
1 subgoal
     *** Declarative Mode ***

  P : nat -> Prop
  x : nat
  HP : P x
  ============================
  thesis :=
   P x
```

In the `assume` variant, the type of the assumed object is mandatory but the name is optional :

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  P : nat -> Prop
  x : nat
  ============================
  thesis :=
   P x -> P x -> P x
```

```
Coq < assume (P x). (* temporary name created *)
1 subgoal
     *** Declarative Mode ***

  P : nat -> Prop
  x : nat
  _hyp : P x
  ============================
  thesis :=
   P x -> P x
```

After `such that`, it is also the case :

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  P : nat -> Prop
  ============================
  thesis :=
   forall x : nat, P x -> P x
```

```
Coq < let x be such that (P x). (* temporary name created *)
1 subgoal
     *** Declarative Mode ***

  P : nat -> Prop
  x : nat
  _hyp : P x
  ============================
  thesis :=
   P x
```

11.3.10 Tuple elimination steps

In the pCIC, many objects dealt with in simple proofs are tuples : pairs , records, existentially quantified formulas. These are so common that the C-zar mathematical proof language provides a mechanism to extract members of those tuples, and also objects in tuples within tuples within tuples...

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  P : nat -> Prop
  A : Prop
  H : exists x : nat, P x /\ A
  ============================
  thesis :=
   A
Coq < consider x such that HP:(P x) and HA:A from H.
1 subgoal
     *** Declarative Mode ***

  P : nat -> Prop
  A : Prop
  H : exists x : nat, P x /\ A
  x : nat
  HP : P x
  HA : A
  ============================
  thesis :=
   A
```

Here is an example with pairs:

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  p : nat * nat
  ============================
  thesis :=
   fst p >= snd p \/ fst p < snd p
Coq < consider x:nat,y:nat from p.
```

```
1 subgoal
     *** Declarative Mode ***

  p : nat * nat
  x : nat
  y : nat
  ============================
  thesis :=
   fst (x, y) >= snd (x, y) \/ fst (x, y) < snd (x, y)
Coq < reconsider thesis as (x >= y \/ x < y).
1 subgoal
     *** Declarative Mode ***

  p : nat * nat
  x : nat
  y : nat
  ============================
  thesis :=
   x >= y \/ x < y
```

It is sometimes desirable to combine assumption and tuple decomposition. This can be done using the `given` command.

```
Coq < Show.
1 subgoal
     *** Declarative Mode ***

  P : nat -> Prop
  HP : forall n : nat, P n -> P (n - 1)
  ============================
  thesis :=
   (exists m : nat, P m) -> P 0
Coq < given m such that Hm:(P m).
1 subgoal
     *** Declarative Mode ***

  P : nat -> Prop
  HP : forall n : nat, P n -> P (n - 1)
  m : nat
  Hm : P m
  ============================
  thesis :=
   P 0
```

11.3.11 Disjunctive reasoning

In some proofs (most of them usually) one has to consider several cases and prove that the `thesis` holds in all the cases. This is done by first specifying which object will be subject to case distinction (usually a disjunction) using `per cases`, and then specifying which case is being proved by using `suppose`.

```
Coq < per cases on HAB.
```

```
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
  C : Prop
  HAC : A -> C
  HBC : B -> C
  HAB : A \/ B
  ============================
  thesis :=
   C
Coq < suppose A.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
  C : Prop
  HAC : A -> C
  HBC : B -> C
  HAB : A \/ B
  _hyp : A
  ============================
  thesis :=
   C
Coq <   hence thesis by HAC.
Subproof completed, now type "end cases" or start a new case.
Coq < suppose HB:B.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
  C : Prop
  HAC : A -> C
  HBC : B -> C
  HAB : A \/ B
  HB : B
  ============================
  thesis :=
   C
Coq <   thus thesis by HB,HBC.
Subproof completed, now type "end cases" or start a new case.
Coq < end cases.
Subproof completed, now type "end proof".
```

The proof is well formed (but incomplete) even if you type `end cases` or the next `suppose` before the previous case is proved.

If the disjunction is derived from a more general principle, e.g. the excluded middle axiom), it is desirable to just specify which instance of it is being used :

```
Coq < Hypothesis EM : forall P:Prop, P \/ ~ P.
EM is assumed

Coq < per cases of (A \/ ~A) by EM.
1 subgoal
     *** Declarative Mode ***

  EM : forall P : Prop, P \/ ~ P
  A : Prop
  C : Prop
  HAC : A -> C
  HNAC : ~ A -> C
  anonymous_matched : A \/ ~ A
  ============================
  thesis :=
   C

Coq < suppose (~A).
1 subgoal
     *** Declarative Mode ***

  EM : forall P : Prop, P \/ ~ P
  A : Prop
  C : Prop
  HAC : A -> C
  HNAC : ~ A -> C
  anonymous_matched : A \/ ~ A
  _hyp : ~ A
  ============================
  thesis :=
   C

Coq <   hence thesis by HNAC.
Subproof completed, now type "end cases" or start a new case.

Coq < suppose A.
1 subgoal
     *** Declarative Mode ***

  EM : forall P : Prop, P \/ ~ P
  A : Prop
  C : Prop
  HAC : A -> C
  HNAC : ~ A -> C
  anonymous_matched : A \/ ~ A
  _hyp : A
  ============================
  thesis :=
   C

Coq <   hence thesis by HAC.
Subproof completed, now type "end cases" or start a new case.

Coq < end cases.
Subproof completed, now type "end proof".
```

11.3.12 Proofs per cases

If the case analysis is to be made on a particular object, the script is very similar: it starts with `per cases on` *object* instead.

```
Coq < per cases on (EM A).
1 subgoal
     *** Declarative Mode ***

  EM : forall P : Prop, P \/ ~ P
  A : Prop
  C : Prop
  HAC : A -> C
  HNAC : ~ A -> C
  ============================
  thesis :=
   C
Coq < suppose (~A).
1 subgoal
     *** Declarative Mode ***

  EM : forall P : Prop, P \/ ~ P
  A : Prop
  C : Prop
  HAC : A -> C
  HNAC : ~ A -> C
  _hyp : ~ A
  ============================
  thesis :=
   C
```

If the object on which a case analysis occurs in the statement to be proved, the command `suppose it is` *pattern* is better suited than `suppose`. *pattern* may contain nested patterns with `as` clauses. A detailed description of patterns is to be found in figure 1.2. here is an example.

```
Coq < per cases on x.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
  x : bool
  ============================
  thesis :=
   (if x then A else B) -> A \/ B
Coq < suppose it is true.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
  x : bool
```

```
  ============================
  thesis :=
   A -> A \/ B

Coq <   assume A.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
  x : bool
  _hyp : A
  ============================
  thesis :=
   A \/ B

Coq <   hence A.
Subproof completed, now type "end cases" or start a new case.

Coq < suppose it is false.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
  x : bool
  ============================
  thesis :=
   B -> A \/ B

Coq <   assume B.
1 subgoal
     *** Declarative Mode ***

  A : Prop
  B : Prop
  x : bool
  _hyp : B
  ============================
  thesis :=
   A \/ B

Coq <   hence B.
Subproof completed, now type "end cases" or start a new case.

Coq < end cases.
Subproof completed, now type "end proof".
```

11.3.13 Proofs by induction

Proofs by induction are very similar to proofs per cases: they start with `per induction on object` and proceed with `suppose it is` *pattern*`and` *induction hypothesis*. The induction hypothesis can be given explicitly or identified by the sub-object m it refers to using `thesis for` *m*.

```
Coq < per induction on n.
1 subgoal
```

```
     *** Declarative Mode ***

  n : nat
  ============================
  thesis :=
   n + 0 = n

Coq < suppose it is 0.
1 subgoal
     *** Declarative Mode ***

  n : nat
  ============================
  thesis :=
   0 + 0 = 0

Coq <   thus (0 + 0 = 0).
Subproof completed, now type "end induction" or start a new case.

Coq < suppose it is (S m) and H:thesis for m.
1 subgoal
     *** Declarative Mode ***

  n : nat
  m : nat
  H : m + 0 = m
  ============================
  thesis :=
   S m + 0 = S m

Coq <   then (S (m + 0) = S m).
1 subgoal
     *** Declarative Mode ***

  n : nat
  m : nat
  H : m + 0 = m
  _fact : S (m + 0) = S m
  ============================
  thesis :=
   S m + 0 = S m

Coq <   thus =~ (S m + 0).
Subproof completed, now type "end induction" or start a new case.

Coq < end induction.
Subproof completed, now type "end proof".
```

11.3.14 Justifications

Intuitively, justifications are hints for the system to understand how to prove the statements the user types in. In the case of this language justifications are made of two components:

Justification objects : `by` followed by a comma-separated list of objects that will be used by a selected tactic to prove the statement. This defaults to the empty list (the statement should then be tautological). The * wildcard provides the usual tactics behavior: use all statements in local context. However, this wildcard should be avoided since it reduces the robustness of the script.

Justification tactic : `using` followed by a Coq tactic that is executed to prove the statement. The default is a solver for (intuitionistic) first-order with equality.

11.4 More details and Formal Semantics

The users looking for more information should have a look at the paper [33]. This paper features a formal semantics of proof state transitions corresponding to the mathematical commands.

Part III

User extensions

Chapter 12

Syntax extensions and interpretation scopes

In this chapter, we introduce advanced commands to modify the way Coq parses and prints objects, i.e. the translations between the concrete and internal representations of terms and commands. The main commands are `Notation` and `Infix` which are described in section 12.1. It also happens that the same symbolic notation is expected in different contexts. To achieve this form of overloading, Coq offers a notion of interpretation scope. This is described in Section 12.2.

Remark: The commands `Grammar`, `Syntax` and `Distfix` which were present for a while in Coq are no longer available from Coq version 8.0. The underlying AST structure is also no longer available. The functionalities of the command `Syntactic Definition` are still available, see Section 12.3.

12.1 Notations

12.1.1 Basic notations

A *notation* is a symbolic abbreviation denoting some term or term pattern.

A typical notation is the use of the infix symbol `/\` to denote the logical conjunction (`and`). Such a notation is declared by

```
Coq < Notation "A /\ B" := (and A B).
```

The expression `(and A B)` is the abbreviated term and the string `"A /\ B"` (called a *notation*) tells how it is symbolically written.

A notation is always surrounded by double quotes (excepted when the abbreviation is a single ident, see 12.3). The notation is composed of *tokens* separated by spaces. Identifiers in the string (such as `A` and `B`) are the *parameters* of the notation. They must occur at least once each in the denoted term. The other elements of the string (such as `/\`) are the *symbols*.

An identifier can be used as a symbol but it must be surrounded by simple quotes to avoid the confusion with a parameter. Similarly, every symbol of at least 3 characters and starting with a simple quote must be quoted (then it starts by two single quotes). Here is an example.

```
Coq < Notation "'IF' c1 'then' c2 'else' c3" := (IF_then_else c1 c2 c3).
```

A notation binds a syntactic expression to a term. Unless the parser and pretty-printer of Coq already know how to deal with the syntactic expression (see 12.1.7), explicit precedences and associativity rules have to be given.

12.1.2 Precedences and associativity

Mixing different symbolic notations in a same text may cause serious parsing ambiguity. To deal with the ambiguity of notations, Coq uses precedence levels ranging from 0 to 100 (plus one extra level numbered 200) and associativity rules.

Consider for example the new notation

```
Coq < Notation "A \/ B" := (or A B).
```

Clearly, an expression such as `forall A:Prop, True /\ A \/ A \/ False` is ambiguous. To tell the Coq parser how to interpret the expression, a priority between the symbols `/\` and `\/` has to be given. Assume for instance that we want conjunction to bind more than disjunction. This is expressed by assigning a precedence level to each notation, knowing that a lower level binds more than a higher level. Hence the level for disjunction must be higher than the level for conjunction.

Since connectives are the less tight articulation points of a text, it is reasonable to choose levels not so far from the higher level which is 100, for example 85 for disjunction and 80 for conjunction[1].

Similarly, an associativity is needed to decide whether `True /\ False /\ False` defaults to `True /\ (False /\ False)` (right associativity) or to `(True /\ False) /\ False` (left associativity). We may even consider that the expression is not well-formed and that parentheses are mandatory (this is a "no associativity")[2]. We don't know of a special convention of the associativity of disjunction and conjunction, let's apply for instance a right associativity (which is the choice of Coq).

Precedence levels and associativity rules of notations have to be given between parentheses in a list of modifiers that the `Notation` command understands. Here is how the previous examples refine.

```
Coq < Notation "A /\ B" := (and A B) (at level 80, right associativity).
Coq < Notation "A \/ B" := (or A B)  (at level 85, right associativity).
```

By default, a notation is considered non associative, but the precedence level is mandatory (except for special cases whose level is canonical). The level is either a number or the mention `next level` whose meaning is obvious. The list of levels already assigned is on Figure 3.1.

12.1.3 Complex notations

Notations can be made from arbitraly complex symbols. One can for instance define prefix notations.

```
Coq < Notation "~ x" := (not x) (at level 75, right associativity).
```

One can also define notations for incomplete terms, with the hole expected to be inferred at typing time.

```
Coq < Notation "x = y" := (@eq _ x y) (at level 70, no associativity).
```

One can define *closed* notations whose both sides are symbols. In this case, the default precedence level for inner subexpression is 200.

```
Coq < Notation "( x , y )" := (@pair _ _ x y) (at level 0).
```

[1] which are the levels effectively chosen in the current implementation of Coq

[2] Coq accepts notations declared as no associative but the parser on which Coq is built, namely CamlP4, currently does not implement the no-associativity and replace it by a left associativity; hence it is the same for Coq: no-associativity is in fact left associativity

One can also define notations for binders.

```
Coq < Notation "{ x : A  |  P }" := (sig A (fun x => P)) (at level 0).
```

In the last case though, there is a conflict with the notation for type casts. This last notation, as shown by the command `Print Grammar constr` is at level 100. To avoid `x : A` being parsed as a type cast, it is necessary to put `x` at a level below 100, typically 99. Hence, a correct definition is

```
Coq < Notation "{ x : A  |  P }" := (sig A (fun x => P)) (at level 0, x at level 99).
```

See the next section for more about factorization.

12.1.4 Simple factorization rules

Coq extensible parsing is performed by Camlp5 which is essentially a LL1 parser. Hence, some care has to be taken not to hide already existing rules by new rules. Some simple left factorization work has to be done. Here is an example.

```
Coq < Notation "x < y"     := (lt x y) (at level 70).
Coq < Notation "x < y < z" := (x < y /\ y < z) (at level 70).
```

In order to factorize the left part of the rules, the subexpression referred by `y` has to be at the same level in both rules. However the default behavior puts `y` at the next level below 70 in the first rule (no associativity is the default), and at the level 200 in the second rule (level 200 is the default for inner expressions). To fix this, we need to force the parsing level of `y`, as follows.

```
Coq < Notation "x < y"     := (lt x y) (at level 70).
Coq < Notation "x < y < z" := (x < y /\ y < z) (at level 70, y at next level).
```

For the sake of factorization with Coq predefined rules, simple rules have to be observed for notations starting with a symbol: e.g. rules starting with "{" or "(" should be put at level 0. The list of Coq predefined notations can be found in Chapter 3.

The command to display the current state of the Coq term parser is

```
Print Grammar constr.
```

Variant:

```
Print Grammar pattern.
```

This displays the state of the subparser of patterns (the parser used in the grammar of the `match with` constructions).

12.1.5 Displaying symbolic notations

The command `Notation` has an effect both on the COQ parser and on the COQ printer. For example:

```
Coq < Check (and True True).
True /\ True
     : Prop
```

However, printing, especially pretty-printing, requires more care than parsing. We may want specific indentations, line breaks, alignment if on several lines, etc.

The default printing of notations is very rudimentary. For printing a notation, a *formatting box* is opened in such a way that if the notation and its arguments cannot fit on a single line, a line break is inserted before the symbols of the notation and the arguments on the next lines are aligned with the argument on the first line.

A first, simple control that a user can have on the printing of a notation is the insertion of spaces at some places of the notation. This is performed by adding extra spaces between the symbols and parameters: each extra space (other than the single space needed to separate the components) is interpreted as a space to be inserted by the printer. Here is an example showing how to add spaces around the bar of the notation.

```
Coq < Notation "{{ x : A  |  P }}" := (sig (fun x : A => P))
Coq <   (at level 0, x at level 99).

Coq < Check (sig (fun x : nat => x=x)).
{{x : nat | x = x}}
     : Set
```

The second, more powerful control on printing is by using the `format` modifier. Here is an example

```
Coq < Notation "'If' c1 'then' c2 'else' c3" := (IF_then_else c1 c2 c3)
Coq < (at level 200, right associativity, format
Coq < "'[v   ' 'If'  c1 '/' '[' 'then'  c2  ']' '/' '[' 'else'  c3 ']' ']'").
Defining 'If' as keyword
```

A *format* is an extension of the string denoting the notation with the possible following elements delimited by single quotes:

- extra spaces are translated into simple spaces
- tokens of the form `'/  '` are translated into breaking point, in case a line break occurs, an indentation of the number of spaces after the "`/`" is applied (2 spaces in the given example)
- token of the form `'//'` force writing on a new line
- well-bracketed pairs of tokens of the form `'[  '` and `']'` are translated into printing boxes; in case a line break occurs, an extra indentation of the number of spaces given after the "`[`" is applied (4 spaces in the example)
- well-bracketed pairs of tokens of the form `'[hv  '` and `']'` are translated into horizontal-orelse-vertical printing boxes; if the content of the box does not fit on a single line, then every breaking point forces a newline and an extra indentation of the number of spaces given after the "`[`" is applied at the beginning of each newline (3 spaces in the example)

- well-bracketed pairs of tokens of the form `'[v '` and `']'` are translated into vertical printing boxes; every breaking point forces a newline, even if the line is large enough to display the whole content of the box, and an extra indentation of the number of spaces given after the "`[`" is applied at the beginning of each newline

Thus, for the previous example, we get

Notations do not survive the end of sections. No typing of the denoted expression is performed at definition time. Type-checking is done only at the time of use of the notation.

```
Coq < Check
Coq <  (IF_then_else (IF_then_else True False True)
Coq <    (IF_then_else True False True)
Coq <    (IF_then_else True False True)).
If If True
      then False
      else True
   then If True
           then False
           else True
   else If True
           then False
           else True
     : Prop
```

Remark: Sometimes, a notation is expected only for the parser. To do so, the option *only parsing* is allowed in the list of modifiers of `Notation`.

12.1.6 The `Infix` command

The `Infix` command is a shortening for declaring notations of infix symbols. Its syntax is

`Infix "`*symbol*`"` `:=` *qualid* `(` *modifier* `,` ... `,` *modifier* `).`

and it is equivalent to

`Notation "x` *symbol* `y"` `:=` `(`*qualid* `x y)` `(` *modifier* `,` ... `,` *modifier* `).`

where `x` and `y` are fresh names distinct from *qualid*. Here is an example.

```
Coq < Infix "/\" := and (at level 80, right associativity).
```

12.1.7 Reserving notations

A given notation may be used in different contexts. Coq expects all uses of the notation to be defined at the same precedence and with the same associativity. To avoid giving the precedence and associativity every time, it is possible to declare a parsing rule in advance without giving its interpretation. Here is an example from the initial state of Coq.

```
Coq < Reserved Notation "x = y" (at level 70, no associativity).
```

Reserving a notation is also useful for simultaneously defined an inductive type or a recursive constant and a notation for it.

Remark: The notations mentioned on Figure 3.1 are reserved. Hence their precedence and associativity cannot be changed.

12.1.8 Simultaneous definition of terms and notations

Thanks to reserved notations, the inductive, coinductive, recursive and corecursive definitions can benefit of customized notations. To do this, insert a `where` notation clause after the definition of the (co)inductive type or (co)recursive term (or after the definition of each of them in case of mutual definitions). The exact syntax is given on Figure 12.1. Here are examples:

```
Coq < Inductive and (A B:Prop) : Prop := conj : A -> B -> A /\ B
Coq < where "A /\ B" := (and A B).

Coq < Fixpoint plus (n m:nat) {struct n} : nat :=
Coq <   match n with
Coq <   | O => m
Coq <   | S p => S (p+m)
Coq <   end
Coq < where "n + m" := (plus n m).
```

12.1.9 Displaying informations about notations

To deactivate the printing of all notations, use the command

```
Unset Printing Notations.
```

To reactivate it, use the command

```
Set Printing Notations.
```

The default is to use notations for printing terms wherever possible.

See also: `Set Printing All` in Section 2.9.

12.1.10 Locating notations

To know to which notations a given symbol belongs to, use the command

`Locate` *symbol*

where symbol is any (composite) symbol surrounded by quotes. To locate a particular notation, use a string where the variables of the notation are replaced by "_".

Example:

```
Coq < Locate "exists".
Notation            Scope
"'exists' x : t , p" := ex (fun x : t => p)
                        : type_scope
                        (default interpretation)
"'exists' x , p" := ex (fun x => p)
                        : type_scope
                        (default interpretation)
"'exists' ! x : A , P" := ex (unique (fun x : A => P))
                        : type_scope
                        (default interpretation)
"'exists' ! x , P" := ex (unique (fun x => P))
                        : type_scope
```

sentence ::= *[*`Local`*]* `Notation` *string* `:=` *term [modifiers] [*`:`*scope]* `.`
| *[*`Local`*]* `Infix` *string* `:=` *qualid [modifiers] [*`:`*scope]* `.`
| *[*`Local`*]* `Reserved Notation` *string [modifiers]* `.`
| `Inductive` *ind_body [decl_notation]* `with` ... `with` *ind_body [decl_notation]*.
| `CoInductive` *ind_body [decl_notation]* `with` ... `with` *ind_body [decl_notation]*.
| `Fixpoint` *fix_body [decl_notation]* `with` ... `with` *fix_body [decl_notation]* `.`
| `CoFixpoint` *cofix_body [decl_notation]* `with` ... `with` *cofix_body [decl_notation]* `.`

decl_notation ::= *[*`where` *string* `:=` *term [*:*scope]]* `.`

modifiers ::= *ident* `,` ... `,` *ident* `at level` *natural*
| *ident* `,` ... `,` *ident* `at next level`
| `at level` *natural*
| `left associativity`
| `right associativity`
| `no associativity`
| *ident* `ident`
| *ident* `global`
| *ident* `bigint`
| `only parsing`
| `format` *string*

Figure 12.1: Syntax of the variants of `Notation`

```
                    (default interpretation)
Coq < Locate "'exists' _ , _".
Notation            Scope
"'exists' x , p" := ex (fun x => p)
                    : type_scope
                    (default interpretation)
```

See also: Section 6.2.9.

12.1.11 Notations with recursive patterns

An experimental mechanism is provided for declaring elementary notations including recursive patterns. The basic syntax is

```
Coq < Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).
```

On the right-hand-side, an extra construction of the form `..` $(f\ t_1 \ldots t_n)$ `..` can be used. Notice that `..` is part of the Coq syntax while ... is just a meta-notation of this manual to denote a sequence of terms of arbitrary size.

This extra construction enclosed within `..`, let's call it t, must be one of the argument of an applicative term of the form $(f\ u_1\ \ldots\ u_n)$. The sequences $t_1 \ldots t_n$ and $u_1 \ldots u_n$ must coincide everywhere but in two places. In one place, say the terms of indice i, we must have $u_i = t$. In the other place, say the terms of indice j, both u_j and t_j must be variables, say x and y which are bound by the notation string on the left-hand-side of the declaration. The variables x and y in the string must occur in a substring of the form "$x\ s$ `..` $s\ y$" where `..` is part of the syntax and s is two times the same sequence of terminal symbols (i.e. symbols which are not variables).

These invariants must be satisfied in order the notation to be correct. The term t_i is the *terminating* expression of the notation and the pattern $(f\ u_1\ \ldots\ u_{i-1}$ [I] $u_{i+1}\ \ldots\ u_{j-1}$ [E] $u_{j+1}\ \ldots\ u_n)$ is the *iterating pattern*. The hole [I] is the *iterative* place and the hole [E] is the *enumerating* place. Remark that if $j < i$, the iterative place comes after the enumerating place accordingly.

The notation parses sequences of tokens such that the subpart "$x\ s$ `..` $s\ y$" parses any number of time (but at least one time) a sequence of expressions separated by the sequence of tokens s. The parsing phase produces a list of expressions which are used to fill in order the holes [E] of the iterating pattern which is nested as many time as the length of the list, the hole [I] being the nesting point. In the innermost occurrence of the nested iterating pattern, the hole [I] is finally filled with the terminating expression.

In the example above, f is `cons`, $n = 3$ (because `cons` has a hidden implicit argument!), $i = 3$ and $j = 2$. The *terminating* expression is `nil` and the *iterating pattern* is `cons` [E] [I]. Finally, the sequence s is made of the single token "`;`". Here is another example.

```
Coq < Notation "( x , y , .. , z )" := (pair .. (pair x y) .. z) (at level 0).
```

Notations with recursive patterns can be reserved like standard notations, they can also be declared within interpretation scopes (see section 12.2).

12.1.12 Notations and binders

Notations can be defined for binders as in the example:

```
Coq < Notation "{ x : A  |  P  }" := (sig (fun x : A => P)) (at level 0).
```

The binding variables in the left-hand-side that occur as a parameter of the notation naturally bind all their occurrences appearing in their respective scope after instantiation of the parameters of the notation.

Contrastingly, the binding variables that are not a parameter of the notation do not capture the variables of same name that could appear in their scope after instantiation of the notation. E.g., for the notation

```
Coq < Notation "'exists_different' n" := (exists p:nat, p<>n) (at level 200).
```

the next command fails because `p` does not bind in the instance of `n`.

```
Coq < Check (exists_different p).
Coq < Coq < Toplevel input, characters 144-145:
> Check (exists_different p).
>                          ^
Error: The reference p was not found in the current environment.
```

Remark: Binding variables must not necessarily be parsed using the `ident` entry. For factorization purposes, they can be said to be parsed at another level (e.g. `x` in "`{ x : A | P }`" must be parsed at level 99 to be factorized with the notation "`{ A } + { B }`" for which `A` can be any term). However, even if parsed as a term, this term must at the end be effectively a single identifier.

12.1.13 Summary

Syntax of notations The different syntactic variants of the command `Notation` are given on Figure 12.1. The optional `:` *scope* is described in the Section 12.2.

Remark: No typing of the denoted expression is performed at definition time. Type-checking is done only at the time of use of the notation.

Remark: Many examples of `Notation` may be found in the files composing the initial state of COQ (see directory `$COQLIB/theories/Init`).

Remark: The notation `"{ x }"` has a special status in such a way that complex notations of the form `"x + { y }"` or `"x * { y }"` can be nested with correct precedences. Especially, every notation involving a pattern of the form `"{ x }"` is parsed as a notation where the pattern `"{ x }"` has been simply replaced by `"x"` and the curly brackets are parsed separately. E.g. `"y + { z }"` is not parsed as a term of the given form but as a term of the form `"y + z"` where `z` has been parsed using the rule parsing `"{ x }"`. Especially, level and precedences for a rule including patterns of the form `"{ x }"` are relative not to the textual notation but to the notation where the curly brackets have been removed (e.g. the level and the associativity given to some notation, say `"{ y } & { z }"` in fact applies to the underlying `"{ x }"`-free rule which is `"y & z"`).

Persistence of notations Notations do not survive the end of sections. They survive modules unless the command `Local Notation` is used instead of `Notation`.

12.2 Interpretation scopes

An *interpretation scope* is a set of notations for terms with their interpretation. Interpretation scopes provides with a weak, purely syntactical form of notations overloading: a same notation, for instance the infix symbol + can be used to denote distinct definitions of an additive operator. Depending on which interpretation scopes is currently open, the interpretation is different. Interpretation scopes can include an interpretation for numerals and strings. However, this is only made possible at the OBJECTIVE CAML level.

See Figure 12.1 for the syntax of notations including the possibility to declare them in a given scope. Here is a typical example which declares the notation for conjunction in the scope `type_scope`.

```
Notation "A /\ B" := (and A B) : type_scope.
```

Remark: A notation not defined in a scope is called a *lonely* notation.

12.2.1 Global interpretation rules for notations

At any time, the interpretation of a notation for term is done within a *stack* of interpretation scopes and lonely notations. In case a notation has several interpretations, the actual interpretation is the one defined by (or in) the more recently declared (or open) lonely notation (or interpretation scope) which defines this notation. Typically if a given notation is defined in some scope *scope* but has also an interpretation not assigned to a scope, then, if *scope* is open before the lonely interpretation is declared, then the lonely interpretation is used (and this is the case even if the interpretation of the notation in *scope* is given after the lonely interpretation: otherwise said, only the order of lonely interpretations and opening of scopes matters, and not the declaration of interpretations within a scope).

The initial state of COQ declares three interpretation scopes and no lonely notations. These scopes, in opening order, are `core_scope`, `type_scope` and `nat_scope`.

The command to add a scope to the interpretation scope stack is

`Open Scope` *scope*.

It is also possible to remove a scope from the interpretation scope stack by using the command

`Close Scope` *scope*.

Notice that this command does not only cancel the last `Open Scope` *scope* but all the invocation of it.

Remark: `Open Scope` and `Close Scope` do not survive the end of sections where they occur. When defined outside of a section, they are exported to the modules that import the module where they occur.

Variants:

1. `Local Open Scope` *scope*.
2. `Local Close Scope` *scope*.

   These variants are not exported to the modules that import the module where they occur, even if outside a section.

12.2.2 Local interpretation rules for notations

In addition to the global rules of interpretation of notations, some ways to change the interpretation of subterms are available.

Local opening of an interpretation scope

It is possible to locally extend the interpretation scope stack using the syntax (*term*)%*key* (or simply *term*%*key* for atomic terms), where *key* is a special identifier called *delimiting key* and bound to a given scope.

In such a situation, the term *term*, and all its subterms, are interpreted in the scope stack extended with the scope bound to *key*.

To bind a delimiting key to a scope, use the command

`Delimit Scope` *scope* `with` *ident*

Binding arguments of a constant to an interpretation scope

It is possible to set in advance that some arguments of a given constant have to be interpreted in a given scope. The command is

`Arguments Scope` *qualid* `[` *opt_scope* ... *opt_scope* `]`

where the list is a list made either of _ or of a scope name. Each scope in the list is bound to the corresponding parameter of *qualid* in order. When interpreting a term, if some of the arguments of *qualid* are built from a notation, then this notation is interpreted in the scope stack extended by the scopes bound (if any) to these arguments.

Variants:

1. `Global Arguments Scope` *qualid* `[` *opt_scope* ... *opt_scope* `]`

   This behaves like `Arguments Scope` *qualid* `[` *opt_scope* ... *opt_scope* `]` but survives when a section is closed instead of stopping working at section closing.

2. `Local Arguments Scope` *qualid* `[` *opt_scope* ... *opt_scope* `]`

   This is a synonym of `Arguments Scope` *qualid* `[` *opt_scope* ... *opt_scope* `]`: if in a section, the effect of the command stops when the section it belongs to ends.

See also: The command to show the scopes bound to the arguments of a function is described in Section 2.

Binding types of arguments to an interpretation scope

When an interpretation scope is naturally associated to a type (e.g. the scope of operations on the natural numbers), it may be convenient to bind it to this type. The effect of this is that any argument of a function that syntactically expects a parameter of this type is interpreted using scope. More precisely, it applies only if this argument is built from a notation, and if so, this notation is interpreted in the scope stack extended by this particular scope. It does not apply to the subterms of this notation (unless the interpretation of the notation itself expects arguments of the same type that would trigger the same scope).

More generally, any *class* (see Chapter 17) can be bound to an interpretation scope. The command to do it is

`Bind Scope` *scope* `with` *class*

Example:

```
Coq < Parameter U : Set.
U is assumed

Coq < Bind Scope U_scope with U.

Coq < Parameter Uplus : U -> U -> U.
Uplus is assumed

Coq < Parameter P : forall T:Set, T -> U -> Prop.
P is assumed

Coq < Parameter f : forall T:Set, T -> U.
f is assumed

Coq < Infix "+" := Uplus : U_scope.

Coq < Unset Printing Notations.

Coq < Open Scope nat_scope. (* Define + on the nat as the default for + *)

Coq < Check (fun x y1 y2 z t => P _ (x + t) ((f _ (y1 + y2) + z))).
fun (x y1 y2 : nat) (z : U) (t : nat) =>
P nat (Peano.plus x t) (Uplus (f nat (Peano.plus y1 y2)) z)
     : nat -> nat -> nat -> U -> nat -> Prop
```

Remark: The scope `type_scope` has also a local effect on interpretation. See the next section.

See also: The command to show the scopes bound to the arguments of a function is described in Section 2.

12.2.3 The `type_scope` interpretation scope

The scope `type_scope` has a special status. It is a primitive interpretation scope which is temporarily activated each time a subterm of an expression is expected to be a type. This includes goals and statements, types of binders, domain and codomain of implication, codomain of products, and more generally any type argument of a declared or defined constant.

12.2.4 Interpretation scopes used in the standard library of COQ

We give an overview of the scopes used in the standard library of COQ. For a complete list of notations in each scope, use the commands `Print Scopes` or `Print Scopes` *scope*.

`type_scope`

This includes infix `*` for product types and infix + for sum types. It is delimited by key `type`.

`nat_scope`

This includes the standard arithmetical operators and relations on type `nat`. Positive numerals in this scope are mapped to their canonical representent built from `O` and `S`. The scope is delimited by key `nat`.

`N_scope`

This includes the standard arithmetical operators and relations on type `N` (binary natural numbers). It is delimited by key `N` and comes with an interpretation for numerals as closed term of type `Z`.

`Z_scope`

This includes the standard arithmetical operators and relations on type `Z` (binary integer numbers). It is delimited by key `Z` and comes with an interpretation for numerals as closed term of type `Z`.

`positive_scope`

This includes the standard arithmetical operators and relations on type `positive` (binary strictly positive numbers). It is delimited by key `positive` and comes with an interpretation for numerals as closed term of type `positive`.

`Q_scope`

This includes the standard arithmetical operators and relations on type `Q` (rational numbers defined as fractions of an integer and a strictly positive integer modulo the equality of the numerator-denominator cross-product). As for numerals, only 0 and 1 have an interpretation in scope `Q_scope` (their interpretations are $\frac{0}{1}$ and $\frac{1}{1}$ respectively).

`Qc_scope`

This includes the standard arithmetical operators and relations on the type `Qc` of rational numbers defined as the type of irreducible fractions of an integer and a strictly positive integer.

`real_scope`

This includes the standard arithmetical operators and relations on type `R` (axiomatic real numbers). It is delimited by key `R` and comes with an interpretation for numerals as term of type `R`. The interpretation is based on the binary decomposition. The numeral 2 is represented by $1 + 1$. The interpretation $\phi(n)$ of an odd positive numerals greater n than 3 is `1+(1+1)*`$\phi((n-1)/2)$. The interpretation $\phi(n)$ of an even positive numerals greater n than 4 is `(1+1)*`$\phi(n/2)$. Negative numerals are represented as the opposite of the interpretation of their absolute value. E.g. the syntactic object `-11` is interpreted as `-(1+(1+1)*((1+1)*(1+(1+1))))` where the unit 1 and all the operations are those of `R`.

`bool_scope`

This includes notations for the boolean operators. It is delimited by key `bool`.

`list_scope`

This includes notations for the list operators. It is delimited by key `list`.

`core_scope`

This includes the notation for pairs. It is delimited by key `core`.

`string_scope`

This includes notation for strings as elements of the type `string`. Special characters and escaping follow Coq conventions on strings (see Section 1.1). Especially, there is no convention to visualize non printable characters of a string. The file `String.v` shows an example that contains quotes, a newline and a beep (i.e. the ascii character of code 7).

`char_scope`

This includes interpretation for all strings of the form `"`c`"` where c is an ascii character, or of the form `"`nnn`"` where nnn is a three-digits number (possibly with leading 0's), or of the form `""""`. Their respective denotations are the ascii code of c, the decimal ascii code nnn, or the ascii code of the character `"` (i.e. the ascii code 34), all of them being represented in the type `ascii`.

12.2.5 Displaying informations about scopes

`Print Visibility`

This displays the current stack of notations in scopes and lonely notations that is used to interpret a notation. The top of the stack is displayed last. Notations in scopes whose interpretation is hidden by the same notation in a more recently open scope are not displayed. Hence each notation is displayed only once.

Variant:

`Print Visibility` *scope*

This displays the current stack of notations in scopes and lonely notations assuming that *scope* is pushed on top of the stack. This is useful to know how a subterm locally occurring in the scope of *scope* is interpreted.

`Print Scope` *scope*

This displays all the notations defined in interpretation scope *scope*. It also displays the delimiting key if any and the class to which the scope is bound, if any.

`Print Scopes`

This displays all the notations, delimiting keys and corresponding class of all the existing interpretation scopes. It also displays the lonely notations.

12.3 Abbreviations

An *abbreviation* is a name, possibly applied to arguments, that denotes a (presumably) more complex expression. Here are examples:

```
Coq < Notation Nlist := (list nat).

Coq < Check 1 :: 2 :: 3 :: nil.
[1; 2; 3]
     : Nlist

Coq < Notation reflexive R := (forall x, R x x).

Coq < Check forall A:Prop, A <-> A.
reflexive iff
     : Prop

Coq < Check reflexive iff.
reflexive iff
     : Prop
```

An abbreviation expects no precedence nor associativity, since it follows the usual syntax of application. Abbreviations are used as much as possible by the Coq printers unless the modifier `(only parsing)` is given.

Abbreviations are bound to an absolute name as an ordinary definition is, and they can be referred by qualified names too.

Abbreviations are syntactic in the sense that they are bound to expressions which are not typed at the time of the definition of the abbreviation but at the time it is used. Especially, abbreviations can be bound to terms with holes (i.e. with "_"). The general syntax for abbreviations is

[`Local`*]* `Notation` *ident [ident ident ... ident ident]* `:=` *term [* `(only parsing)` *] .*

Example:

```
Coq < Definition explicit_id (A:Set) (a:A) := a.
explicit_id is defined

Coq < Notation id := (explicit_id _).

Coq < Check (id 0).
id 0
     : nat
```

Abbreviations do not survive the end of sections. No typing of the denoted expression is performed at definition time. Type-checking is done only at the time of use of the abbreviation.

12.4 Tactic Notations

Tactic notations allow to customize the syntax of the tactics of the tactic language[3]. Tactic notations obey the following syntax

sentence	::=	`Tactic Notation` *[tactic_level] production_item ... production_item* `:=` *tactic* `.`
production_item	::=	*string* | *tactic_argument_type* `(` *ident* `)`
tactic_level	::=	`(at level` *natural* `)`
tactic_argument_type	::=	`ident` | `simple_intropattern` | `reference`
	|	`hyp` | `hyp_list` | `ne_hyp_list`
	|	`constr` | `constr_list` | `ne_constr_list`
	|	`integer` | `integer_list` | `ne_integer_list`
	|	`int_or_var` | `int_or_var_list` | `ne_int_or_var_list`
	|	`tactic` | `tactic`n (for $0 \leq n \leq 5$)

A tactic notation `Tactic Notation` *tactic_level [production_item ... production_item]* `:=` *tactic* extends the parser and pretty-printer of tactics with a new rule made of the list of production items. It then evaluates into the tactic expression *tactic*. For simple tactics, it is recommended to use a terminal symbol, i.e. a *string*, for the first production item. The tactic level indicates the parsing precedence of the tactic notation. This information is particularly relevant for notations of tacticals. Levels 0 to 5 are available (default is 0). To know the parsing precedences of the existing tacticals, use the command `Print Grammar tactic.`

Each type of tactic argument has a specific semantic regarding how it is parsed and how it is interpreted. The semantic is described in the following table. The last command gives examples of tactics which use the corresponding kind of argument.

Tactic argument type	parsed as	interpreted as	as in tactic
`ident`	identifier	a user-given name	`intro`
`simple_intropattern`	intro_pattern	an intro_pattern	`intros`
`hyp`	identifier	an hypothesis defined in context	`clear`
`reference`	qualified identifier	a global reference of term	`unfold`
`constr`	term	a term	`exact`
`integer`	integer	an integer	
`int_or_var`	identifier or integer	an integer	`do`
`tactic`	tactic at level 5	a tactic	
`tactic`n	tactic at level n	a tactic	
entry`_list`	list of *entry*	a list of how *entry* is interpreted	
`ne_`*entry*`_list`	non-empty list of *entry*	a list of how *entry* is interpreted	

Remark: In order to be bound in tactic definitions, each syntactic entry for argument type must include the case of simple $\mathcal{L}_{tac}$ identifier as part of what it parses. This is naturally the case for `ident`, `simple_intropattern`, `reference`, `constr`, ... but not for `integer`. This is the reason for introducing a special entry `int_or_var` which evaluates to integers only but which syntactically includes identifiers in order to be usable in tactic definitions.

[3] Tactic notations are just a simplification of the `Grammar tactic simple_tactic` command that existed in versions prior to version 8.0.

Remark: The *entry*`_list` and `ne_`*entry*`_list` entries can be used in primitive tactics or in other notations at places where a list of the underlying entry can be used: *entry* is either `constr`, `hyp`, `integer` or `int_or_var`.

Part IV

Practical tools

Chapter 13

The Coq commands

There are three Coq commands:

- `coqtop`: The Coq toplevel (interactive mode) ;
- `coqc` : The Coq compiler (batch compilation).
- `coqchk` : The Coq checker (validation of compiled libraries)

The options are (basically) the same for the first two commands, and roughly described below. You can also look at the `man` pages of `coqtop` and `coqc` for more details.

13.1 Interactive use (`coqtop`)

In the interactive mode, also known as the Coq toplevel, the user can develop his theories and proofs step by step. The Coq toplevel is run by the command `coqtop`.

They are two different binary images of Coq: the byte-code one and the native-code one (if Objective Caml provides a native-code compiler for your platform, which is supposed in the following). When invoking `coqtop` or `coqc`, the native-code version of the system is used. The command-line options `-byte` and `-opt` explicitly select the byte-code and the native-code versions, respectively.

The byte-code toplevel is based on a Caml toplevel (to allow the dynamic link of tactics). You can switch to the Caml toplevel with the command `Drop.`, and come back to the Coq toplevel with the command `Toplevel.loop();;`.

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 6.4).

Warning: The name *file* must be a regular Coq identifier, as defined in the Section 1.1. It must only contain letters, digits or underscores (_). Thus it can be `/bar/foo/toto.v` but cannot be `/bar/foo/to-to.v`.

Notice that the `-byte` and `-opt` options are still available with `coqc` and allow you to select the byte-code or native-code versions of the system.

13.3 Resource file

When Coq is launched, with either `coqtop` or `coqc`, the resource file `$HOME/.coqrc.7.0` is loaded, where `$HOME` is the home directory of the user. If this file is not found, then the file `$HOME/.coqrc` is searched. 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`).

This file may contain, for instance, `Add LoadPath` commands to add directories to the load path of Coq. It is possible to skip the loading of the resource file with the option `-q`.

13.4 Environment variables

There are three environment variables used by the Coq system. `$COQBIN` for the directory where the binaries are, `$COQLIB` for the directory where the standard library is, and `$COQTOP` for the directory of the sources. The latter is useful only for developers that are writing their own tactics and are using `coq_makefile` (see 14.3). If `$COQBIN` or `$COQLIB` are not defined, Coq will use the default values (defined at installation time). So these variables are useful only if you move the Coq binaries and library after installation.

13.5 Options

The following command-line options are recognized by the commands `coqc` and `coqtop`, unless stated otherwise:

`-byte`

Run the byte-code version of Coq.

`-opt`

Run the native-code version of Coq.

`-I` ***directory***, `-include` ***directory***

Add physical path *directory* to the list of directories where to look for a file and bind it to the empty logical directory. The subdirectory structure of *directory* is recursively available from Coq using absolute names (see Section 2.6.2).

`-I` ***directory*** `-as` ***dirpath***

Add physical path *directory* to the list of directories where to look for a file and bind it to the logical directory *dirpath*. The subdirectory structure of *directory* is recursively available from Coq using absolute names extending the *dirpath* prefix.

See also: `Add LoadPath` in Section 6.5.3 and logical paths in Section 2.6.1.

`-R` ***directory dirpath***, `-R` ***directory*** `-as` ***dirpath***

Do as `-I` *directory* `-as` *dirpath* but make the subdirectory structure of *directory* recursively visible so that the recursive contents of physical *directory* is available from Coq using short or partially qualified names.

See also: `Add Rec LoadPath` in Section 6.5.4 and logical paths in Section 2.6.1.

`-top` ***dirpath***

This sets the toplevel module name to *dirpath* instead of `Top`. Not valid for `coqc`.

`-notop` ***dirpath***

This sets the toplevel module name to the empty logical dirpath. Not valid for `coqc`.

`-exclude-dir` ***subdirectory***

This tells to exclude any subdirectory named *subdirectory* while processing option `-R`. Without this option only the conventional version control management subdirectories named `CVS` and `_darcs` are excluded.

`-is` ***file*****,** `-inputstate` ***file***

Cause Coq to use the state put in the file *file* as its input state. The default state is *initial.coq*. Mainly useful to build the standard input state.

`-outputstate` ***file***

Cause Coq to dump its state to file *file*.coq just after finishing parsing and evaluating all the arguments from the command line.

`-nois`

Cause Coq to begin with an empty state. Mainly useful to build the standard input state.

`-init-file` ***file***

Take *file* as the resource file.

`-q`

Cause Coq not to load the resource file.

`-user` ***username***

Take resource file of user *username* (that is *~username*`/.coqrc.7.0`) instead of yours.

`-load-ml-source` ***file***

Load the Caml source file *file*.

`-load-ml-object` ***file***

Load the Caml object file *file*.

`-l` ***file*****,** `-load-vernac-source` ***file***

Load Coq file *file*`.v`

`-lv` ***file*****,** `-load-vernac-source-verbose` ***file***

Load Coq file *file*`.v` with a copy of the contents of the file on standard input.

`-load-vernac-object` ***file***

Load Coq compiled file *file*`.vo`

`-require` ***file***

Load Coq compiled file *file*`.vo` and import it (`Require` *file*).

`-compile` ***file***

This compiles file *file*`.v` into *file*`.vo`. This option implies options `-batch` and `-silent`. It is only available for `coqtop`.

`-compile-verbose` ***file***

This compiles file *file*`.v` into *file*`.vo` with a copy of the contents of the file on standard input. This option implies options `-batch` and `-silent`. It is only available for `coqtop`.

`-verbose`

This option is only for `coqc`. It tells to compile the file with a copy of its contents on standard input.

`-batch`

Batch mode : exit just after arguments parsing. This option is only used by `coqc`.

`-xml`

This option is for use with `coqc`. It tells Coq to export on the standard output the content of the compiled file into XML format.

`-quality` Improve the legibility of the proof terms produced by some tactics.

`-emacs`

Tells Coq it is executed under Emacs.

`-impredicative-set`

Change the logical theory of Coq by declaring the sort `Set` impredicative; warning: this is known to be inconsistent with some standard axioms of classical mathematics such as the functional axiom of choice or the principle of description

`-dump-glob` ***file***

This dumps references for global names in file *file* (to be used by coqdoc, see 14.4)

`-dont-load-proofs`

This avoids loading in memory the proofs of opaque theorems resulting in a smaller memory requirement and faster compilation; warning: this invalidates some features such as the extraction tool.

`-vm`

This activates the use of the bytecode-based conversion algorithm for the current session (see Section 6.9.4).

`-image` ***file***

This option sets the binary image to be used to be *file* instead of the standard one. Not of general use.

`-bindir` ***directory***

Set for `coqc` the directory containing Coq binaries. It is equivalent to do `export COQBIN=`*directory* before lauching `coqc`.

`-where`

Print the Coq's standard library location and exit.

`-v`

Print the Coq's version and exit.

`-h`, `-help`

Print a short usage and exit.

13.6 Compiled libraries checker (coqchk)

The `coqchk` command takes a list of library paths as argument. The corresponding compiled libraries (.vo files) are searched in the path, recursively processing the libraries they depend on. The content of all these libraries is then type-checked. The effect of `coqchk` is only to return with normal exit code in case of success, and with positive exit code if an error has been found. Error messages are not deemed to help the user understand what is wrong. In the current version, it does not modify the compiled libraries to mark them as succesfully checked.

Note that non-logical information is not checked. By logical information, we mean the type and optional body associated to names. It excludes for instance anything related to the concrete syntax of objects (customized syntax rules, association between short and long names), implicit arguments, etc.

This tool can be used for several purposes. One is to check that a compiled library provided by a third-party has not been forged and that loading it cannot introduce inconsistencies.[1] Another point is to get an even higher level of security. Since `coqtop` can be extended with custom tactics, possibly ill-typed code, it cannot be guaranteed that the produced compiled libraries are correct. `coqchk` is a standalone verifier, and thus it cannot be tainted by such malicious code.

Command-line options `-I`, `-R`, `-where` and `-impredicative-set` are supported by `coqchk` and have the same meaning as for `coqtop`. Extra options are:

`-norec` *module*

Check *module* but do not force check of its dependencies.

`-admit` *module*

Do not check *module* and any of its dependencies, unless explicitely required.

`-o`

At exit, print a summary about the context. List the names of all assumptions and variables (constants without body).

`-silent`

Do not write progress information in standard output.

Environment variable `$COQLIB` can be set to override the location of the standard library.

The algorithm for deciding which modules are checked or admitted is the following: assuming that `coqchk` is called with argument M, option `-norec` N, and `-admit` A. Let us write $\overline{S}$ the set of reflexive transitive dependencies of set S. Then:

[1] Ill-formed non-logical information might for instance bind `Coq.Init.Logic.True` to short name `False`, so apprently `False` is inhabited, but using fully qualified names, `Coq.Init.Logic.False` will always refer to the absurd proposition, what we guarantee is that there is no proof of this latter constant.

- Modules $C = \overline{M} \backslash \overline{A} \cup M \cup N$ are loaded and type-checked before being added to the context.
- And $\overline{M} \cup \overline{N} \backslash C$ is the set of modules that are loaded and added to the context without type-checking. Basic integrity checks (checksums) are nonetheless performed.

As a rule of thumb, the `-admit` can be used to tell that some libraries have already been checked. So `coqchk A B` can be split in `coqchk A && coqchk B -admit A` without type-checking any definition twice. Of course, the latter is slightly slower since it makes more disk access. It is also less secure since an attacker might have replaced the compiled library A after it has been read by the first command, but before it has been read by the second command.

Chapter 14

Utilities

The distribution provides utilities to simplify some tedious works beside proof development, tactics writing or documentation.

14.1 Building a toplevel extended with user tactics

The native-code version of Coq cannot dynamically load user tactics using Objective Caml code. It is possible to build a toplevel of Coq, with Objective Caml code statically linked, with the tool `coqmktop`.

For example, one can build a native-code Coq toplevel extended with a tactic which source is in `tactic.ml` with the command

```
% coqmktop -opt -o mytop.out tactic.cmx
```

where `tactic.ml` has been compiled with the native-code compiler `ocamlopt`. This command generates an executable called `mytop.out`. To use this executable to compile your Coq files, use `coqc -image mytop.out`.

A basic example is the native-code version of Coq (`coqtop.opt`), which can be generated by `coqmktop -opt -o coqopt.opt`.

Application: how to use the Objective Caml debugger with Coq. One useful application of `coqmktop` is to build a Coq toplevel in order to debug your tactics with the Objective Caml debugger. You need to have configured and compiled Coq for debugging (see the file `INSTALL` included in the distribution). Then, you must compile the Caml modules of your tactic with the option `-g` (with the bytecode compiler) and build a stand-alone bytecode toplevel with the following command:

```
% coqmktop -g -o coq-debug <your .cmo files>
```

To launch the Objective Caml debugger with the image you need to execute it in an environment which correctly sets the `COQLIB` variable. Moreover, you have to indicate the directories in which `ocamldebug` should search for Caml modules.

A possible solution is to use a wrapper around `ocamldebug` which detects the executables containing the word `coq`. In this case, the debugger is called with the required additional arguments. In other cases, the debugger is simply called without additional arguments. Such a wrapper can be found in the `dev/` subdirectory of the sources.

14.2 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 OBJECTIVE CAML 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`), but also at the command `Declare ML Module`.

Dependencies of OBJECTIVE CAML modules are computed by looking at `open` commands and the dot notation *module.value*. However, this is done approximatively and you are advised to use `ocamldep` instead for the OBJECTIVE CAML modules dependencies.

See the man page of `coqdep` for more details and options.

14.3 Creating a `Makefile` for COQ modules

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 be a tedious work and that's why COQ provides a tool to automate its creation, `coq_makefile`. Given the files to compile, the command `coq_makefile` prints a `Makefile` on the standard output. So one has just to run the command:

```
% coq_makefile file1.v ... filen.v > Makefile
```

The resulted `Makefile` has a target `depend` which computes the dependencies and puts them in a separate file `.depend`, which is included by the `Makefile`. Therefore, you should create such a file before the first invocation of make. You can for instance use the command

```
% touch .depend
```

Then, to initialize or update the modules dependencies, type in:

```
% make depend
```

There is a target `all` to compile all the files $file_1 \ldots file_n$, 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`).

`coq_makefile` can also handle the case of ML files and subdirectories. For more options type

```
% coq_makefile -help
```

Warning: To compile a project containing OBJECTIVE CAML files you must keep the sources of COQ somewhere and have an environment variable named `COQTOP` that points to that directory.

14.4 Documenting COQ files with coqdoc

coqdoc is a documentation tool for the proof assistant COQ, similar to javadoc or ocamldoc. The task of coqdoc is

1. to produce a nice LaTeX and/or HTML document from the COQ sources, readable for a human and not only for the proof assistant;
2. to help the user navigating in his own (or third-party) sources.

14.4.1 Principles

Documentation is inserted into COQ files as *special comments*. Thus your files will compile as usual, whether you use coqdoc or not. coqdoc presupposes that the given COQ files are well-formed (at least lexically). Documentation starts with `(**`, followed by a space, and ends with the pending `*)`. The documentation format is inspired by Todd A. Coram's *Almost Free Text (AFT)* tool: it is mainly ASCII text with some syntax-light controls, described below. coqdoc is robust: it shouldn't fail, whatever the input is. But remember: "garbage in, garbage out".

COQ material inside documentation. COQ material is quoted between the delimiters `[` and `]`. Square brackets may be nested, the inner ones being understood as being part of the quoted code (thus you can quote a term like $[x : T]u$ by writing `[[x:T]u]`). Inside quotations, the code is pretty-printed in the same way as it is in code parts.

Pre-formatted vernacular is enclosed by `[[` and `]]`. The former must be followed by a newline and the latter must follow a newline.

Pretty-printing. coqdoc uses different faces for identifiers and keywords. The pretty-printing of COQ tokens (identifiers or symbols) can be controlled using one of the following commands:

```
(** printing token %...LaTeX...% #...HTML...# *)
```

or

```
(** printing token $...LaTeX math...$ #...HTML...# *)
```

It gives the LaTeX and HTML texts to be produced for the given COQ token. One of the LaTeX or HTML text may be ommitted, causing the default pretty-printing to be used for this token.

The printing for one token can be removed with

```
(** remove printing token *)
```

Initially, the pretty-printing table contains the following mapping:

`->`	$\rightarrow$	`<-`	$\leftarrow$	`*`	$\times$
`<=`	$\leq$	`>=`	$\geq$	`=>`	$\Rightarrow$
`<>`	$\neq$	`<->`	$\leftrightarrow$	`|-`	$\vdash$
`\/`	$\vee$	`/\`	$\wedge$	`~`	$\neg$

Any of these can be overwritten or suppressed using the `printing` commands.

Important note: the recognition of tokens is done by a (ocaml)lex automaton and thus applies the longest-match rule. For instance, `->~` is recognized as a single token, where COQ sees two tokens. It is the responsability of the user to insert space between tokens *or* to give pretty-printing rules for the possible combinations, e.g.

```
(** printing ->~ %\ensuremath{\rightarrow\lnot}% *)
```

Sections. Sections are introduced by 1 to 4 leading stars (i.e. at the beginning of the line) followed by a space. One star is a section, two stars a sub-section, etc. The section title is given on the remaining of the line. Example:

```
(** * Well-founded relations

    In this section, we introduce...  *)
```

Lists. List items are introduced by 1 to 4 leading dashes. Deepness of the list is indicated by the number of dashes. List ends with a blank line. Example:

```
This module defines
    - the predecessor [pred]
    - the addition [plus]
    - order relations:
      -- less or equal [le]
      -- less [lt]
```

Rules. More than 4 leading dashes produce an horizontal rule.

Escapings to LaTeX and HTML. Pure LaTeX or HTML material can be inserted using the following escape sequences:

- `$...LaTeX stuff...$` inserts some LaTeX material in math mode. Simply discarded in HTML output.
- `%...LaTeX stuff...%` inserts some LaTeX material. Simply discarded in HTML output.
- `#...HTML stuff...#` inserts some HTML material. Simply discarded in LaTeX output.

Verbatim. Verbatim material is introduced by a leading << and closed by >> at the beginning of a line. Example:

```
Here is the corresponding caml code:
<<
  let rec fact n =
    if n <= 1 then 1 else n * fact (n-1)
>>
```

Hyperlinks. Hyperlinks can be inserted into the HTML output, so that any identifier is linked to the place of its definition.

In order to get hyperlinks you need to first compile your Coq file using `coqc --dump-glob` *`file`*; this appends Coq names resolutions done during the compilation to file *`file`*. Take care of erasing this file, if any, when starting the whole compilation process.

Then invoke `coqdoc --glob-from` *`file`* to tell coqdoc to look for name resolutions into the file *`file`*.

Identifiers from the Coq standard library are linked to the Coq web site at `http://coq.inria.fr/library/`. This behavior can be changed using command line options `--no-externals` and `--coqlib`; see below.

Hiding / Showing parts of the source. Some parts of the source can be hidden using command line options `-g` and `-l` (see below), or using such comments:

```
(* begin hide *)
some Coq material
(* end hide *)
```

Conversely, some parts of the source which would be hidden can be shown using such comments:

```
(* begin show *)
some Coq material
(* end show *)
```

The latter cannot be used around some inner parts of a proof, but can be used around a whole proof.

14.4.2 Usage

coqdoc is invoked on a shell command line as follows:

`coqdoc` < *options and files* >

Any command line argument which is not an option is considered to be a file (even if it starts with a `-`). COQ files are identified by the suffixes `.v` and `.g` and LaTeX files by the suffix `.tex`.

HTML output

This is the default output. One HTML file is created for each COQ file given on the command line, together with a file `index.html` (unless option `-no-index` is passed). The HTML pages use a style sheet named `style.css`. Such a file is distributed with coqdoc.

LaTeX output

A single LaTeX file is created, on standard output. It can be redirected to a file with option `-o`. The order of files on the command line is kept in the final document. LaTeX files given on the command line are copied 'as is' in the final document . DVI and PostScript can be produced directly with the options `-dvi` and `-ps` respectively.

TeXmacs output

To translate the input files to TeXmacs format, to be used by the TeXmacs Coq interface (see http://www-sop.inria.fr/lemme/Philippe.Audebaud/tmcoq/).

Command line options

Overall options

`--html`

Select a HTML output.

`--latex`

Select a LaTeX output.

--dvi

Select a DVI output.

--ps

Select a PostScript output.

--texmacs

Select a TEXmacs output.

-stdout

Write output to stdout.

-o ***file*****, --output** ***file***

Redirect the output into the file '*file*' (meaningless with `-html`).

-d ***dir*****, --directory** ***dir***

Output files into directory '*dir*' instead of current directory (option `-d` does not change the filename specified with option `-o`, if any).

-s , --short

Do not insert titles for the files. The default behavior is to insert a title like "Library Foo" for each file.

-t ***string*****, --title** ***string***

Set the document title.

--body-only

Suppress the header and trailer of the final document. Thus, you can insert the resulting document into a larger one.

-p ***string*****, --preamble** ***string***

Insert some material in the LATEX preamble, right before `\begin{document}` (meaningless with `-html`).

--vernac-file ***file*****, --tex-file** ***file***

Considers the file '*file*' respectively as a `.v` (or `.g`) file or a `.tex` file.

--files-from ***file***

Read file names to process in file '*file*' as if they were given on the command line. Useful for program sources splitted in several directories.

-q, --quiet

Be quiet. Do not print anything except errors.

-h, --help

Give a short summary of the options and exit.

-v, --version

Print the version and exit.

Index options

Default behavior is to build an index, for the HTML output only, into `index.html`.

`--no-index`

Do not output the index.

`--multi-index`

Generate one page for each category and each letter in the index, together with a top page `index.html`.

Table of contents option

`-toc, --table-of-contents`

Insert a table of contents. For a LaTeX output, it inserts a `\tableofcontents` at the beginning of the document. For a HTML output, it builds a table of contents into `toc.html`.

Hyperlinks options

`--glob-from` *file*

Make references using COQ globalizations from file *file*. (Such globalizations are obtained with COQ option `-dump-glob`).

`--no-externals`

Do not insert links to the COQ standard library.

`--coqlib` *url*

Set base URL for the COQ standard library (default is `http://coq.inria.fr/library/`).

`-R` *dir coqdir*

Map physical directory *dir* to COQ logical directory *coqdir* (similarly to COQ option `-R`).

Note: option `-R` only has effect on the files *following* it on the command line, so you will probably need to put this option first.

Contents options

`-g, --gallina`

Do not print proofs.

`-l, --light`

Light mode. Suppress proofs (as with `-g`) and the following commands:

- [`Recursive`] `Tactic Definition`
- `Hint / Hints`
- `Require`
- `Transparent / Opaque`

- `Implicit Argument / Implicits`
- `Section / Variable / Hypothesis / End`

The behavior of options `-g` and `-l` can be locally overridden using the `(* begin show *) ... (* end show *)` environment (see above).

Language options

Default behavior is to assume ASCII 7 bits input files.

`-latin1`, `--latin1`

Select ISO-8859-1 input files. It is equivalent to `-inputenc latin1 -charset iso-8859-1`.

`-utf8`, `--utf8`

Select UTF-8 (Unicode) input files. It is equivalent to `-inputenc utf8 -charset utf-8`. LaTeX UTF-8 support can be found at http://www.ctan.org/tex-archive/macros/latex/contrib/supported/unicode/.

`--inputenc` *string*

Give a LaTeX input encoding, as an option to LaTeX package `inputenc`.

`--charset` *string*

Specify the HTML character set, to be inserted in the HTML header.

14.4.3 The coqdoc LaTeX style file

In case you choose to produce a document without the default LaTeX preamble (by using option `--no-preamble`), then you must insert into your own preamble the command

```
\usepackage{coqdoc}
```

Then you may alter the rendering of the document by redefining some macros:

`coqdockw`, `coqdocid`

The one-argument macros for typesetting keywords and identifiers. Defaults are sans-serif for keywords and italic for identifiers.

For example, if you would like a slanted font for keywords, you may insert

```
\renewcommand{\coqdockw}[1]{\textsl{#1}}
```

anywhere between `\usepackage{coqdoc}` and `\begin{document}`.

`coqdocmodule`

One-argument macro for typesetting the title of a `.v` file. Default is

```
\newcommand{\coqdocmodule}[1]{\section*{Module #1}}
```

and you may redefine it using `\renewcommand`.

14.5 Exporting Coq theories to XML

This section describes the exportation of Coq theories to XML that has been contributed by Claudio Sacerdoti Coen. Currently, the main applications are the rendering and searching tool developed within the HELM[1] and MoWGLI[2] projects mainly at the University of Bologna and partly at INRIA-Sophia Antipolis.

14.5.1 Practical use of the XML exportation tool

The basic way to export the logical content of a file into XML format is to use `coqc` with option `-xml`. When the `-xml` flag is set, every definition or declaration is immediately exported to XML once concluded. The system environment variable `COQ_XML_LIBRARY_ROOT` must be previously set to a directory in which the logical structure of the exported objects is reflected.

For `Makefile` files generated by `coq_makefile` (see section 14.3), it is sufficient to compile the files using

```
make COQ_XML=-xml
```

(or, equivalently, setting the environment variable `COQ_XML`)

To export a development to XML, the suggested procedure is then:

1. add to your own contribution a valid `Make` file and use `coq_makefile` to generate the `Makefile` from the `Make` file.

   Warning: Since logical names are used to structure the XML hierarchy, always add to the `Make` file at least one `"-R"` option to map physical file names to logical module paths.

2. set the `COQ_XML_LIBRARY_ROOT` environment variable to the directory where the XML file hierarchy must be physically rooted.

3. compile your contribution with `"make COQ_XML=-xml"`

Remark: In case the system variable `COQ_XML_LIBRARY_ROOT` is not set, the output is done on the standard output. Also, the files are compressed using `gzip` after creation. This is to save disk space since the XML format is very verbose.

14.5.2 Reflection of the logical structure into the file system

For each Coq logical object, several independent files associated to this object are created. The structure of the long name of the object is reflected in the directory structure of the file system. E.g. an object of long name $ident_1 . \ldots . ident_n . ident$ is exported to files in the subdirectory $ident_1/\ldots/ident_n$ of the directory bound to the environment variable `COQ_XML_LIBRARY_ROOT`.

[1] Hypertextual Electronic Library of Mathematics
[2] Mathematics on the Web, Get it by Logic and Interfaces

14.5.3 What is exported?

The XML exportation tool exports the logical content of Coq theories. This covers global definitions (including lemmas, theorems, ...), global assumptions (parameters and axioms), local assumptions or definitions, and inductive definitions.

Vernacular files are exported to `.theory.xml` files. Comments are pre-processed with coqdoc (see section 14.4). Especially, they have to be enclosed within `(**` and `*)` to be exported.

For each inductive definition of name $ident_1 \ldots ident_n.ident$, a file named *ident*`.ind.xml` is created in the subdirectory $ident_1/\ldots/ident_n$ of the xml library root directory. It contains the arities and constructors of the type. For mutual inductive definitions, the file is named after the name of the first inductive type of the block.

For each global definition of base name $ident_1 \ldots ident_n.ident$, files named *ident*`.con.body.xml` and *ident*`.con.xml` are created in the subdirectory $ident_1/\ldots/ident_n$. They respectively contain the body and the type of the definition.

For each global assumption of base name $ident_1.ident_2 \ldots ident_n.ident$, a file named *ident*`.con.xml` is created in the subdirectory $ident_1/\ldots/ident_n$. It contains the type of the global assumption.

For each local assumption or definition of base name *ident* located in sections $ident'_1, \ldots, ident'_p$ of the module $ident_1.ident_2 \ldots ident_n.ident$, a file named *ident*`.var.xml` is created in the subdirectory $ident_1/\ldots/ident_n/ident'_1/\ldots/ident'_p$. It contains its type and, if a definition, its body.

In order to do proof-rendering (for example in natural language), some redundant typing information is required, i.e. the type of at least some of the subterms of the bodies and types of the CIC objects. These types are called inner types and are exported to files of suffix `.types.xml` by the exportation tool.

14.5.4 Inner types

The type of a subterm of a construction is called an *inner type* if it respects the following conditions.

1. Its sort is `Prop`[3].
2. It is not a type cast nor an atomic term (variable, constructor or constant).
3. If it's root is an abstraction, then the root's parent node is not an abstraction, i.e. only the type of the outer abstraction of a block of nested abstractions is printed.

The rationale for the 3^{rd} condition is that the type of the inner abstractions could be easily computed starting from the type of the outer ones; moreover, the types of the inner abstractions requires a lot of disk/memory space: removing the 3^{rd} condition leads to XML file that are two times as big as the ones exported applying the 3^{rd} condition.

14.5.5 Interactive exportation commands

There are also commands to be used interactively in `coqtop`.

[3] or `CProp` which is the "sort"-like definition used in C-CoRN (see http://vacuumcleaner.cs.kun.nl/c-corn) to type computationally relevant predicative propositions.

`Print XML` *qualid*

If the variable `COQ_XML_LIBRARY_ROOT` is set, this command creates files containing the logical content in XML format of *qualid*. If the variable is not set, the result is displayed on the standard output.

Variants:

1. `Print XML File` *string* *qualid*
   This writes the logical content of *qualid* in XML format to files whose prefix is *string*.

`Show XML Proof`

If the variable `COQ_XML_LIBRARY_ROOT` is set, this command creates files containing the current proof in progress in XML format. It writes also an XML file made of inner types. If the variable is not set, the result is displayed on the standard output.

Variants:

1. `Show XML File` *string* `Proof`
   This writes the logical content of *qualid* in XML format to files whose prefix is *string*.

14.5.6 Applications: rendering, searching and publishing

The HELM team at the University of Bologna has developed tools exploiting the XML exportation of Coq libraries. This covers rendering, searching and publishing tools.

All these tools require a running http server and, if possible, a MathML compliant browser. The procedure to install the suite of tools ultimately allowing rendering and searching can be found on the HELM web site http://helm.cs.unibo.it/library.html.

It may be easier though to upload your developments on the HELM http server and to re-use the infrastructure running on it. This requires publishing your development. To this aim, follow the instructions on http://mowgli.cs.unibo.it.

Notice that the HELM server already hosts a copy of the standard library of Coq and of the Coq user contributions.

14.5.7 Technical informations

CIC with Explicit Named Substitutions

The exported files are XML encoding of the lambda-terms used by the Coq system. The implementative details of the Coq system are hidden as much as possible, so that the XML DTD is a straightforward encoding of the Calculus of (Co)Inductive Constructions.

Nevertheless, there is a feature of the Coq system that can not be hidden in a completely satisfactory way: discharging (see Sect.2.4). In Coq it is possible to open a section, declare variables and use them in the rest of the section as if they were axiom declarations. Once the section is closed, every definition and theorem in the section is discharged by abstracting it over the section variables. Variable declarations as well as section declarations are entirely dropped. Since we are interested in an XML encoding of definitions and theorems as close as possible to those directly provided the user, we do not want to export discharged forms. Exporting non-discharged theorem and definitions together with theorems that rely on the discharged forms obliges the tools that work on the XML encoding to implement discharging to achieve logical consistency. Moreover, the rendering of the files can be misleading, since hyperlinks can

be shown between occurrences of the discharge form of a definition and the non-discharged definition, that are different objects.

To overcome the previous limitations, Claudio Sacerdoti Coen developed in his PhD. thesis an extension of CIC, called Calculus of (Co)Inductive Constructions with Explicit Named Substitutions, that is a slight extension of CIC where discharging is not necessary. The DTD of the exported XML files describes constants, inductive types and variables of the Calculus of (Co)Inductive Constructions with Explicit Named Substitutions. The conversion to the new calculus is performed during the exportation phase.

The following example shows a very small COQ development together with its version in CIC with Explicit Named Substitutions.

```
# CIC version: #
Section S.
 Variable A : Prop.

 Definition impl := A -> A.

 Theorem t : impl.           (* uses the undischarged form of impl *)
  Proof.
   exact (fun (a:A) => a).
  Qed.

End S.

Theorem t' : (impl False).   (* uses the discharged form of impl *)
 Proof.
  exact (t False).           (* uses the discharged form of t *)
 Qed.

# Corresponding CIC with Explicit Named Substitutions version: #
Section S.
 Variable A : Prop.

 Definition impl(A) := A -> A. (* theorems and definitions are
                                  explicitly abstracted over the
                                  variables. The name is sufficient to
                                  completely describe the abstraction *)

 Theorem t(A) : impl.          (* impl where A is not instantiated *)
  Proof.
   exact (fun (a:A) => a).
  Qed.

End S.

Theorem t'() : impl{False/A}. (* impl where A is instantiated with False
                                 Notice that t' does not depend on A     *)
 Proof.
```

```
  exact t{False/A}.           (* t where A is instantiated with False *)
Qed.
```

Further details on the typing and reduction rules of the calculus can be found in Claudio Sacerdoti Coen PhD. dissertation, where the consistency of the calculus is also proved.

The CIC with Explicit Named Substitutions XML DTD

A copy of the DTD can be found in the file "cic.dtd" in the contrib/xml source directory of COQ. The following is a very brief overview of the elements described in the DTD.

<ConstantType> is the root element of the files that correspond to constant types.

<ConstantBody> is the root element of the files that correspond to constant bodies. It is used only for closed definitions and theorems (i.e. when no metavariable occurs in the body or type of the constant)

<CurrentProof> is the root element of the file that correspond to the body of a constant that depends on metavariables (e.g. unfinished proofs)

<Variable> is the root element of the files that correspond to variables

<InductiveTypes> is the root element of the files that correspond to blocks of mutually defined inductive definitions

The elements <LAMBDA>, <CAST>, <PROD>, <REL>, <SORT>, <APPLY>, <VAR>, <META>, <IMPLICIT>, <CONST>, <LETIN>, <MUTIND>, <MUTCONSTRUCT>, <MUTCASE>, <FIX> and <COFIX> are used to encode the constructors of CIC. The sort or type attribute of the element, if present, is respectively the sort or the type of the term, that is a sort because of the typing rules of CIC.

The element <instantiate> correspond to the application of an explicit named substitution to its first argument, that is a reference to a definition or declaration in the environment.

All the other elements are just syntactic sugar.

14.6 Embedded COQ phrases inside LATEX documents

When writing a documentation about a proof development, one may want to insert COQ phrases inside a LATEX document, possibly together with the corresponding answers of the system. We provide a mechanical way to process such 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 named *file*.v.tex with the COQ outcome.

There are options to produce the COQ parts in smaller font, italic, between horizontal rules, etc. See the man page of coq-tex for more details.

Remark. This Reference Manual and the Tutorial have been completely produced with coq-tex.

14.7 COQ and GNU EMACS

14.7.1 The COQ Emacs mode

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 in the style of the Caml GNU EMACS mode.

Add the following lines to your `.emacs` file:

```
(setq auto-mode-alist (cons '("\\.v$" . coq-mode) auto-mode-alist))
(autoload 'coq-mode "coq" "Major mode for editing Coq vernacular." t)
```

The COQ major mode is triggered by visiting a file with extension `.v`, or manually with the command `M-x coq-mode`. It gives you the correct syntax table for the COQ language, and also a rudimentary indentation facility:

- pressing TAB at the beginning of a line indents the line like the line above;
- extra TABs increase the indentation level (by 2 spaces by default);
- M-TAB decreases the indentation level.

An inferior mode to run COQ under Emacs, by Marco Maggesi, is also included in the distribution, in file `coq-inferior.el`. Instructions to use it are contained in this file.

14.7.2 Proof General

Proof General is a generic interface for proof assistants based on Emacs (or XEmacs). The main idea is that the COQ commands you are editing are sent to a COQ toplevel running behind Emacs and the answers of the system automatically inserted into other Emacs buffers. Thus you don't need to copy-paste the COQ material from your files to the COQ toplevel or conversely from the COQ toplevel to some files.

Proof General is developped and distributed independently of the system COQ. It is freely available at `proofgeneral.inf.ed.ac.uk`.

14.8 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 *file*`.v` gives birth to the specification file *file*`.g` (where the suffix `.g` stands for GALLINA).

See the man page of `gallina` for more details and options.

14.9 Man pages

There are man pages for the commands `coqdep`, `gallina` and `coq-tex`. Man pages are installed at installation time (see installation instructions in file `INSTALL`, step 6).

Chapter 15

CoqIDE Integrated Development Environment

The CoqIDE Integrated Development Environment is a graphical tool, to be used as a user-friendly replacement to `coqtop`. Its main purpose is to allow the user to navigate forward and backward into a Coq vernacular file, executing corresponding commands or undoing them respectively.

CoqIDE is run by typing the command `coqide` on the command line. Without argument, the main screen is displayed with an "unnamed buffer", and with a file name as argument, another buffer displaying the contents of that file. Additionally, `coqide` accepts the same options as `coqtop`, given in Chapter 13, the ones having obviously no meaning for CoqIDE being ignored. Additionally, `coqide` accepts the option `-enable-geoproof` to enable the support for *GeoProof* [1].

A sample CoqIDE main screen, while navigating into a file `Fermat.v`, is shown on Figure 15.1. At the top is a menu bar, and a tool bar below it. The large window on the left is displaying the various *script buffers*. The upper right window is the *goal window*, where goals to prove are displayed. The lower right window is the *message window*, where various messages resulting from commands are displayed. At the bottom is the status bar.

15.1 Managing files and buffers, basic edition

In the script window, you may open arbitrarily many buffers to edit. The *File* menu allows you to open files or create some, save them, print or export them into various formats. Among all these buffers, there is always one which is the current *running buffer*, whose name is displayed on a green background, which is the one where Coq commands are currently executed.

Buffers may be edited as in any text editor, and classical basic editing commands (Copy/Paste, ...) are available in the *Edit* menu. CoqIDE offers only basic editing commands, so if you need more complex editing commands, you may launch your favorite text editor on the current buffer, using the *Edit/External Editor* menu.

[1] *GeoProof* is dynamic geometry software which can be used in conjunction with CoqIDE to interactively build a Coq statement corresponding to a geometric figure. More information about *GeoProof* can be found here: `http://home.gna.org/geoproof/`

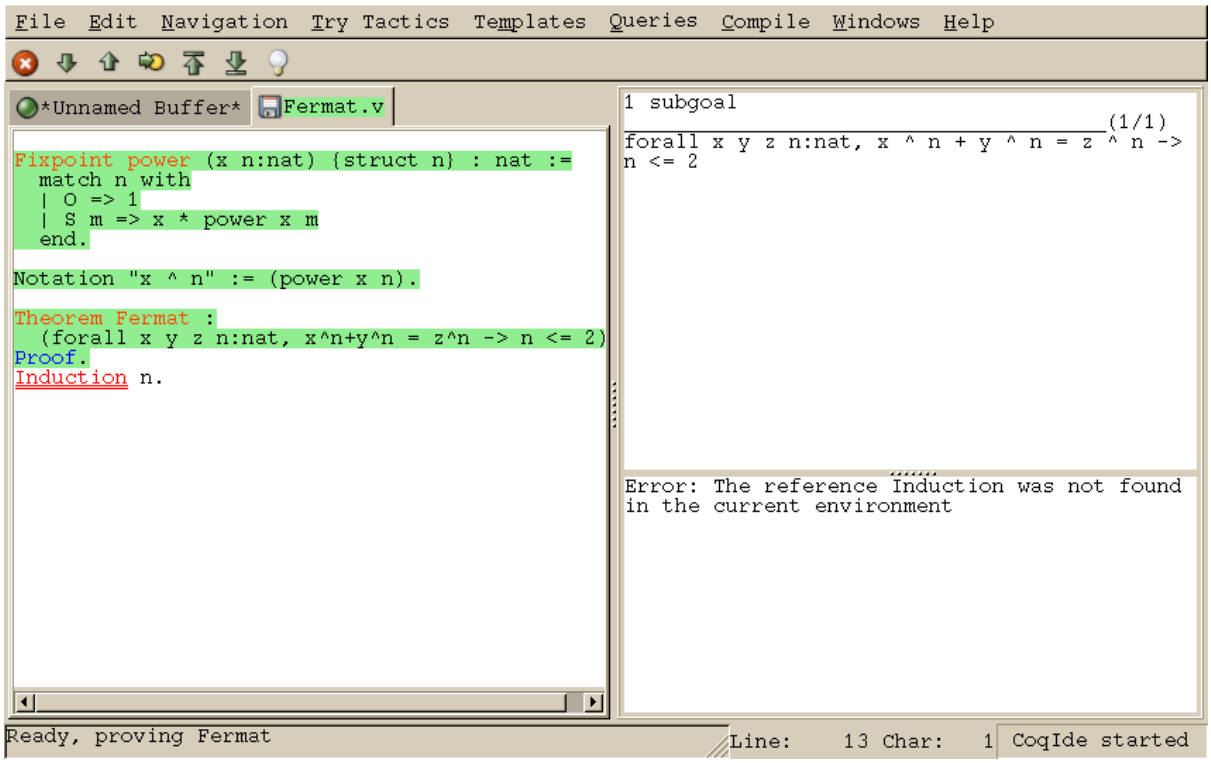

Figure 15.1: CoqIDE main screen

15.2 Interactive navigation into Coq scripts

The running buffer is the one where navigation takes place. The toolbar proposes five basic commands for this. The first one, represented by a down arrow icon, is for going forward executing one command. If that command is successful, the part of the script that has been executed is displayed on a green background. If that command fails, the error message is displayed in the message window, and the location of the error is emphasized by a red underline.

On Figure 15.1, the running buffer is `Fermat.v`, all commands until the `Theorem` have been already executed, and the user tried to go forward executing `Induction n`. That command failed because no such tactic exist (tactics are now in lowercase...), and the wrong word is underlined.

Notice that the green part of the running buffer is not editable. If you ever want to modify something you have to go backward using the up arrow tool, or even better, put the cursor where you want to go back and use the goto button. Unlike with `coqtop`, you should never use `Undo` to go backward.

Two additional tool buttons exist, one to go directly to the end and one to go back to the beginning. If you try to go to the end, or in general to run several commands using the goto button, the execution will stop whenever an error is found.

If you ever try to execute a command which happens to run during a long time, and would like to abort it before its termination, you may use the interrupt button (the white cross on a red circle).

Finally, notice that these navigation buttons are also available in the menu, where their keyboard shortcuts are given.

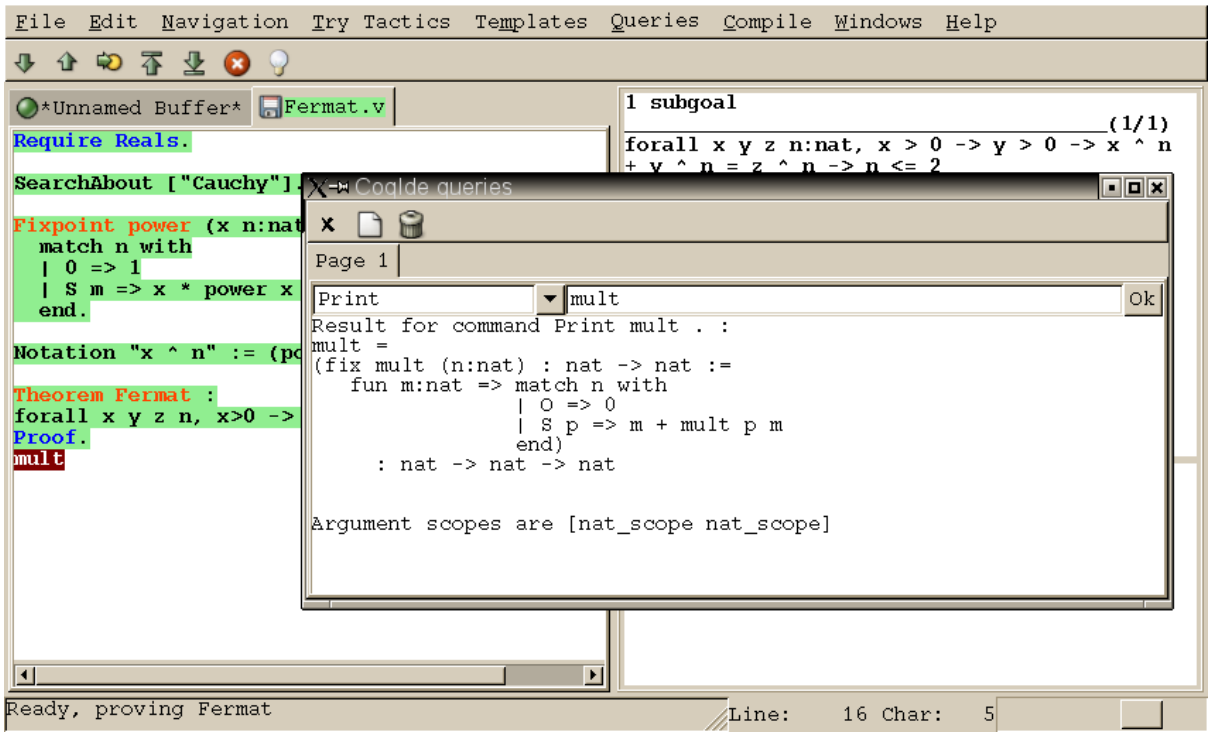

Figure 15.2: CoqIDE: the query window

15.3 Try tactics automatically

The menu `Try Tactics` provides some features for automatically trying to solve the current goal using simple tactics. If such a tactic succeeds in solving the goal, then its text is automatically inserted into the script. There is finally a combination of these tactics, called the *proof wizard* which will try each of them in turn. This wizard is also available as a tool button (the light bulb). The set of tactics tried by the wizard is customizable in the preferences.

These tactics are general ones, in particular they do not refer to particular hypotheses. You may also try specific tactics related to the goal or one of the hypotheses, by clicking with the right mouse button one the goal or the considered hypothesis. This is the "contextual menu on goals" feature, that may be disabled in the preferences if undesirable.

15.4 Vernacular commands, templates

The `Templates` menu allows to use shortcuts to insert vernacular commands. This is a nice way to proceed if you are not sure of the spelling of the command you want.

Moreover, this menu offers some *templates* which will automatic insert a complex command like Fixpoint with a convenient shape for its arguments.

15.5 Queries

We call *query* any vernacular command that do not change the current state, such as `Check`, `SearchAbout`, etc. Those commands are of course useless during compilation of a file, hence should not be included in scripts. To run such commands without writing them in the script, COQIDE offers another input window called the *query window*. This window can be displayed on demand, either by using the `Window` menu, or directly using shortcuts given in the `Queries` menu. Indeed, with COQIDE the simplest way to perform a `SearchAbout` on some identifier is to select it using the mouse, and pressing `F2`. This will both make appear the query window and run the `SearchAbout` in it, displaying the result. Shortcuts `F3` and `F4` are for `Check` and `Print` respectively. Figure 15.2 displays the query window after selection of the word "mult" in the script windows, and pressing `F4` to print its definition.

15.6 Compilation

The `Compile` menu offers direct commands to:

- compile the current buffer
- run a compilation using `make`
- go to the last compilation error
- create a `makefile` using `coq_makefile`.

15.7 Customizations

You may customize your environment using menu `Edit/Preferences`. A new window will be displayed, with several customization sections presented as a notebook.

The first section is for selecting the text font used for scripts, goal and message windows.

The second section is devoted to file management: you may configure automatic saving of files, by periodically saving the contents into files named `#f#` for each opened file `f`. You may also activate the *revert* feature: in case a opened file is modified on the disk by a third party, COQIDE may read it again for you. Note that in the case you edited that same file, you will be prompt to choose to either discard your changes or not. The `File charset encoding` choice is described below in Section 15.8.3

The `Externals` section allows to customize the external commands for compilation, printing, web browsing. In the browser command, you may use `%s` to denote the URL to open, for example: `mozilla -remote "OpenURL(%s)"`.

The `Tactics Wizard` section allows to defined the set of tactics that should be tried, in sequence, to solve the current goal.

The last section is for miscellaneous boolean settings, such as the "contextual menu on goals" feature presented in Section 15.3.

Notice that these settings are saved in the file `.coqiderc` of your home directory.

A gtk2 accelerator keymap is saved under the name `.coqide.keys`. This file should not be edited manually: to modify a given menu shortcut, go to the corresponding menu item without releasing the mouse button, press the key you want for the new shortcut, and release the mouse button afterwards.

For experts: it is also possible to set up a specific gtk resource file, under the name `.coqide-gtk2rc`, following the gtk2 resources syntax http://developer.gnome.org/doc/API/2.0/gtk/gtk-Resource-Files.html. Such a default resource file can be found

in the subdirectory `lib/coq/ide` of the root installation directory of Coq (alternatively, it can be found in the subdirectory `ide` of the source archive of Coq). You may copy this file into your home directory, and edit it using any text editor, CoqIDE itself for example.

15.8 Using unicode symbols

CoqIDE supports unicode character encoding in its text windows, consequently a large set of symbols is available for notations.

15.8.1 Displaying unicode symbols

You just need to define suitable notations as described in Chapter 12. For example, to use the mathematical symbols ∀ and ∃, you may define

```
Notation "∀ x :  t, P" :=
(forall x:t, P) (at level 200, x ident).
Notation "∃ x :  t, P" :=
(exists x:t, P) (at level 200, x ident).
```

There exists a small set of such notations already defined, in the file `utf8.v` of Coq library, so you may enable them just by `Require utf8` inside CoqIDE, or equivalently, by starting CoqIDE with `coqide -l utf8`.

However, there are some issues when using such unicode symbols: you of course need to use a character font which supports them. In the Fonts section of the preferences, the Preview line displays some unicode symbols, so you could figure out if the selected font is OK. Related to this, one thing you may need to do is choose whether Gtk should use antialiased fonts or not, by setting the environment variable `GDK_USE_XFT` to 1 or 0 respectively.

15.8.2 Defining an input method for non ASCII symbols

To input an Unicode symbol, a general method is to press both the CONTROL and the SHIFT keys, and type the hexadecimal code of the symbol required, for example `2200` for the ∀ symbol. A list of symbol codes is available at `http://www.unicode.org`.

Of course, this method is painful for symbols you use often. There is always the possibility to copy-paste a symbol already typed in. Another method is to bind some key combinations for frequently used symbols. For example, to bind keys `F11` and `F12` to ∀ and ∃ respectively, you may add

```
bind "F11" "insert-at-cursor" ("∀")
bind "F12" "insert-at-cursor" ("∃")
```

to your `binding "text"` section in `.coqide-gtk2rc`.

15.8.3 Character encoding for saved files

In the `Files` section of the preferences, the encoding option is related to the way files are saved.

If you have no need to exchange files with non UTF-8 aware applications, it is better to choose the UTF-8 encoding, since it guarantees that your files will be read again without problems. (This is because when CoqIDE reads a file, it tries to automatically detect its character encoding.)

If you choose something else than UTF-8, then missing characters will be written encoded by `\x{....}` or `\x{........}` where each dot is an hexadecimal digit: the number between braces is the hexadecimal UNICODE index for the missing character.

15.9 Building a custom COQIDE with user ML code

You can do this as described in Section 14.1 for a custom coq text toplevel, simply by adding option `-ide` to `coqmktop`, that is something like

```
coqmktop -ide -byte m1.cmo ... mn.cmo
```

or

```
coqmktop -ide -opt m1.cmx ... mn.cmx
```

Part V

Addendum to the Reference Manual

Presentation of the Addendum

Here you will find several pieces of additional documentation for the Coq Reference Manual. Each of this chapters is concentrated on a particular topic, that should interest only a fraction of the Coq users: that's the reason why they are apart from the Reference Manual.

Extended pattern-matching This chapter details the use of generalized pattern-matching. It is contributed by Cristina Cornes and Hugo Herbelin.

Implicit coercions This chapter details the use of the coercion mechanism. It is contributed by Amokrane Saïbi.

Program extraction This chapter explains how to extract in practice ML files from F_ω terms. It is contributed by Jean-Christophe Filliâtre and Pierre Letouzey.

Program This chapter explains the use of the `Program` vernacular which allows the development of certified programs in Coq. It is contributed by Matthieu Sozeau and replaces the previous `Program` tactic by Catherine Parent.

omega `omega`, written by Pierre Crégut, solves a whole class of arithmetic problems.

The `ring` tactic This is a tactic to do AC rewriting. This chapter explains how to use it and how it works. The chapter is contributed by Patrick Loiseleur.

The `Setoid_replace` tactic This is a tactic to do rewriting on types equipped with specific (only partially substitutive) equality. The chapter is contributed by Clément Renard.

Calling external provers This chapter describes several tactics which call external provers.

Contents

Chapter 16

Extended pattern-matching

Cristina Cornes and Hugo Herbelin

This section describes the full form of pattern-matching in Coq terms.

16.1 Patterns

The full syntax of `match` is presented in Figures 1.1 and 1.2. Identifiers in patterns are either constructor names or variables. Any identifier that is not the constructor of an inductive or coinductive type is considered to be a variable. A variable name cannot occur more than once in a given pattern. It is recommended to start variable names by a lowercase letter.

If a pattern has the form $(c\ \vec{x})$ where c is a constructor symbol and $\vec{x}$ is a linear vector of (distinct) variables, it is called *simple*: it is the kind of pattern recognized by the basic version of `match`. On the opposite, if it is a variable x or has the form $(c\ \vec{p})$ with p not only made of variables, the pattern is called *nested*.

A variable pattern matches any value, and the identifier is bound to that value. The pattern "_" (called "don't care" or "wildcard" symbol) also matches any value, but does not bind anything. It may occur an arbitrary number of times in a pattern. Alias patterns written `(`*pattern* `as` *identifier*`)` are also accepted. This pattern matches the same values as *pattern* does and *identifier* is bound to the matched value. A pattern of the form *pattern* | *pattern* is called disjunctive. A list of patterns separated with commas is also considered as a pattern and is called *multiple pattern*. However multiple patterns can only occur at the root of pattern-matching equations. Disjunctions of *multiple pattern* are allowed though.

Since extended `match` expressions are compiled into the primitive ones, the expressiveness of the theory remains the same. Once the stage of parsing has finished only simple patterns remain. Re-nesting of pattern is performed at printing time. An easy way to see the result of the expansion is to toggle off the nesting performed at printing (use here `Set Printing Matching`), then by printing the term with `Print` if the term is a constant, or using the command `Check`.

The extended `match` still accepts an optional *elimination predicate* given after the keyword `return`. Given a pattern matching expression, if all the right-hand-sides of => (*rhs* in short) have the same type, then this type can be sometimes synthesized, and so we can omit the `return` part. Otherwise the predicate after `return` has to be provided, like for the basic `match`.

Let us illustrate through examples the different aspects of extended pattern matching. Consider for example the function that computes the maximum of two natural numbers. We can write it in primitive syntax by:

```
Coq < Fixpoint max (n m:nat) {struct m} : nat :=
Coq <   match n with
Coq <   | O => m
Coq <   | S n' => match m with
Coq <             | O => S n'
Coq <             | S m' => S (max n' m')
Coq <             end
Coq <   end.
max is recursively defined (decreasing on 2nd argument)
```

Multiple patterns Using multiple patterns in the definition of `max` allows to write:

```
Coq < Reset max.

Coq < Fixpoint max (n m:nat) {struct m} : nat :=
Coq <   match n, m with
Coq <   | O, _ => m
Coq <   | S n', O => S n'
Coq <   | S n', S m' => S (max n' m')
Coq <   end.
max is recursively defined (decreasing on 2nd argument)
```

which will be compiled into the previous form.

The pattern-matching compilation strategy examines patterns from left to right. A `match` expression is generated **only** when there is at least one constructor in the column of patterns. E.g. the following example does not build a `match` expression.

```
Coq < Check (fun x:nat => match x return nat with
Coq <                     | y => y
Coq <                     end).
fun x : nat => x
     : nat -> nat
```

Aliasing subpatterns We can also use "`as` *ident*" to associate a name to a sub-pattern:

```
Coq < Reset max.

Coq < Fixpoint max (n m:nat) {struct n} : nat :=
Coq <   match n, m with
Coq <   | O, _ => m
Coq <   | S n' as p, O => p
Coq <   | S n', S m' => S (max n' m')
Coq <   end.
max is recursively defined (decreasing on 1st argument)
```

Nested patterns Here is now an example of nested patterns:

```
Coq < Fixpoint even (n:nat) : bool :=
Coq <   match n with
Coq <   | O => true
Coq <   | S O => false
Coq <   | S (S n') => even n'
Coq <   end.
even is recursively defined (decreasing on 1st argument)
```

This is compiled into:

```
Coq < Print even.
even =
fix even (n : nat) : bool :=
  match n with
  | 0 => true
  | 1 => false
  | S (S n') => even n'
  end
     : nat -> bool
Argument scope is [nat_scope]
```

In the previous examples patterns do not conflict with, but sometimes it is comfortable to write patterns that admit a non trivial superposition. Consider the boolean function `lef` that given two natural numbers yields `true` if the first one is less or equal than the second one and `false` otherwise. We can write it as follows:

```
Coq < Fixpoint lef (n m:nat) {struct m} : bool :=
Coq <   match n, m with
Coq <   | O, x => true
Coq <   | x, O => false
Coq <   | S n, S m => lef n m
Coq <   end.
lef is recursively defined (decreasing on 2nd argument)
```

Note that the first and the second multiple pattern superpose because the couple of values `O O` matches both. Thus, what is the result of the function on those values? To eliminate ambiguity we use the *textual priority rule*: we consider patterns ordered from top to bottom, then a value is matched by the pattern at the ith row if and only if it is not matched by some pattern of a previous row. Thus in the example, `O O` is matched by the first pattern, and so `(lef O O)` yields `true`.

Another way to write this function is:

```
Coq < Reset lef.

Coq < Fixpoint lef (n m:nat) {struct m} : bool :=
Coq <   match n, m with
Coq <   | O, x => true
Coq <   | S n, S m => lef n m
Coq <   | _, _ => false
Coq <   end.
lef is recursively defined (decreasing on 2nd argument)
```

Here the last pattern superposes with the first two. Because of the priority rule, the last pattern will be used only for values that do not match neither the first nor the second one.

Terms with useless patterns are not accepted by the system. Here is an example:

```
Coq < Check (fun x:nat =>
Coq <          match x with
Coq <          | O => true
Coq <          | S _ => false
Coq <          | x => true
Coq <          end).
Coq < Coq < Toplevel input, characters 246-255:
>           | x => true
>             ^^^^^^^^^
Error: This clause is redundant.
```

Disjunctive patterns Multiple patterns that share the same right-hand-side can be factorized using the notation *mult_pattern* `|` ... `|` *mult_pattern*. For instance, `max` can be rewritten as follows:

```
Coq < Fixpoint max (n m:nat) {struct m} : nat :=
Coq <   match n, m with
Coq <   | S n', S m' => S (max n' m')
Coq <   | 0, p | p, 0 => p
Coq <   end.
max is recursively defined (decreasing on 2nd argument)
```

Similarly, factorization of (non necessary multiple) patterns that share the same variables is possible by using the notation *pattern* `|` ... `|` *pattern*. Here is an example:

```
Coq < Definition filter_2_4 (n:nat) : nat :=
Coq <   match n with
Coq <   | 2 as m | 4 as m => m
Coq <   | _ => 0
Coq <   end.
filter_2_4 is defined
```

Here is another example using disjunctive subpatterns.

```
Coq < Definition filter_some_square_corners (p:nat*nat) : nat*nat :=
Coq <   match p with
Coq <   | ((2 as m | 4 as m), (3 as n | 5 as n)) => (m,n)
Coq <   | _ => (0,0)
Coq <   end.
filter_some_square_corners is defined
```

16.2 About patterns of parametric types

When matching objects of a parametric type, constructors in patterns *do not expect* the parameter arguments. Their value is deduced during expansion. Consider for example the type of polymorphic lists:

```
Coq < Inductive List (A:Set) : Set :=
Coq <   | nil : List A
Coq <   | cons : A -> List A -> List A.
List is defined
List_rect is defined
List_ind is defined
List_rec is defined
```

We can check the function *tail*:

```
Coq < Check
Coq <   (fun l:List nat =>
Coq <      match l with
Coq <      | nil => nil nat
Coq <      | cons _ l' => l'
Coq <      end).
fun l : List nat => match l with
                    | nil => nil nat
                    | cons _ l' => l'
                    end
     : List nat -> List nat
```

When we use parameters in patterns there is an error message:

```
Coq < Check
Coq <   (fun l:List nat =>
Coq <      match l with
Coq <      | nil A => nil nat
Coq <      | cons A _ l' => l'
Coq <      end).
Coq < Coq < Toplevel input, characters 196-201:
>      | nil A => nil nat
>        ^^^^^
Error: The constructor nil expects no arguments.
```

16.3 Matching objects of dependent types

The previous examples illustrate pattern matching on objects of non-dependent types, but we can also use the expansion strategy to destructure objects of dependent type. Consider the type `listn` of lists of a certain length:

```
Coq < Inductive listn : nat -> Set :=
Coq <   | niln : listn 0
Coq <   | consn : forall n:nat, nat -> listn n -> listn (S n).
listn is defined
listn_rect is defined
listn_ind is defined
listn_rec is defined
```

16.3.1 Understanding dependencies in patterns

We can define the function `length` over `listn` by:

```
Coq < Definition length (n:nat) (l:listn n) := n.
length is defined
```

Just for illustrating pattern matching, we can define it by case analysis:

```
Coq < Reset length.
Coq < Definition length (n:nat) (l:listn n) :=
Coq <   match l with
Coq <   | niln => 0
Coq <   | consn n _ _ => S n
Coq <   end.
length is defined
```

We can understand the meaning of this definition using the same notions of usual pattern matching.

16.3.2 When the elimination predicate must be provided

The examples given so far do not need an explicit elimination predicate because all the rhs have the same type and the strategy succeeds to synthesize it. Unfortunately when dealing with dependent patterns it often happens that we need to write cases where the type of the rhs are different instances of the elimination predicate. The function `concat` for `listn` is an example where the branches have different type and we need to provide the elimination predicate:

```
Coq < Fixpoint concat (n:nat) (l:listn n) (m:nat) (l':listn m) {struct l} :
Coq <  listn (n + m) :=
Coq <   match l in listn n return listn (n + m) with
Coq <   | niln => l'
Coq <   | consn n' a y => consn (n' + m) a (concat n' y m l')
Coq <   end.
concat is recursively defined (decreasing on 2nd argument)
```

The elimination predicate is `fun (n:nat) (l:listn n) => listn (n+m)`. In general if m has type $(I\ q_1 \ldots q_r\ t_1 \ldots t_s)$ where $q_1 \ldots q_r$ are parameters, the elimination predicate should be of the form : `fun` $y_1 \ldots y_s$ x`: (`$I\ q_1 \ldots q_r\ y_1 \ldots y_s$`) => Q`.

In the concrete syntax, it should be written :

match m as x in $(I \_ \ldots \_\ y_1 \ldots y_s)$ return Q with $\ldots$ end

The variables which appear in the in and as clause are new and bounded in the property Q in the return clause. The parameters of the inductive definitions should not be mentioned and are replaced by _.

Recall that a list of patterns is also a pattern. So, when we destructure several terms at the same time and the branches have different type we need to provide the elimination predicate for this multiple pattern. It is done using the same scheme, each term may be associated to an as and in clause in order to introduce a dependent product.

For example, an equivalent definition for `concat` (even though the matching on the second term is trivial) would have been:

```
Coq < Reset concat.
Coq < Fixpoint concat (n:nat) (l:listn n) (m:nat) (l':listn m) {struct l} :
```

```
Coq <  listn (n + m) :=
Coq <   match l in listn n, l' return listn (n + m) with
Coq <   | niln, x => x
Coq <   | consn n' a y, x => consn (n' + m) a (concat n' y m x)
Coq <   end.
concat is recursively defined (decreasing on 2nd argument)
```

When the arity of the predicate (i.e. number of abstractions) is not correct Coq raises an error message. For example:

```
Coq < Fixpoint concat
Coq <  (n:nat) (l:listn n) (m:nat)
Coq <  (l':listn m) {struct l} : listn (n + m) :=
Coq <   match l, l' with
Coq <   | niln, x => x
Coq <   | consn n' a y, x => consn (n' + m) a (concat n' y m x)
Coq <   end.
Coq < Coq < Coq < Toplevel input, characters 342-343:
>   | niln, x => x
>                ^
Error:
In environment
concat : forall n : nat,
         listn n -> forall m : nat, listn m -> listn (n + m)
n : nat
l : listn n
m : nat
l' : listn m
The term "l'" has type "listn m" while it is expected to have type
 "listn (?61 + ?62)".
```

16.4 Using pattern matching to write proofs

In all the previous examples the elimination predicate does not depend on the object(s) matched. But it may depend and the typical case is when we write a proof by induction or a function that yields an object of dependent type. An example of proof using `match` in given in Section 10.1.

For example, we can write the function `buildlist` that given a natural number n builds a list of length n containing zeros as follows:

```
Coq < Fixpoint buildlist (n:nat) : listn n :=
Coq <   match n return listn n with
Coq <   | O => niln
Coq <   | S n => consn n 0 (buildlist n)
Coq <   end.
buildlist is recursively defined (decreasing on 1st argument)
```

We can also use multiple patterns. Consider the following definition of the predicate less-equal `Le`:

```
Coq < Inductive LE : nat -> nat -> Prop :=
Coq <   | LEO : forall n:nat, LE 0 n
Coq <   | LES : forall n m:nat, LE n m -> LE (S n) (S m).
LE is defined
LE_ind is defined
```

We can use multiple patterns to write the proof of the lemma `forall (n m:nat), (LE n m)\/(LE m n)`:

```
Coq < Fixpoint dec (n m:nat) {struct n} : LE n m \/ LE m n :=
Coq <   match n, m return LE n m \/ LE m n with
Coq <   | O, x => or_introl (LE x 0) (LEO x)
Coq <   | x, O => or_intror (LE x 0) (LEO x)
Coq <   | S n as n', S m as m' =>
Coq <       match dec n m with
Coq <       | or_introl h => or_introl (LE m' n') (LES n m h)
Coq <       | or_intror h => or_intror (LE n' m') (LES m n h)
Coq <       end
Coq <   end.
dec is recursively defined (decreasing on 1st argument)
```

In the example of `dec`, the first `match` is dependent while the second is not.

The user can also use `match` in combination with the tactic `refine` (see Section 8.2.2) to build incomplete proofs beginning with a `match` construction.

16.5 Pattern-matching on inductive objects involving local definitions

If local definitions occur in the type of a constructor, then there are two ways to match on this constructor. Either the local definitions are skipped and matching is done only on the true arguments of the constructors, or the bindings for local definitions can also be caught in the matching.

Example.

```
Coq < Inductive list : nat -> Set :=
Coq <   | nil : list 0
Coq <   | cons : forall n:nat, let m := (2 * n) in list m -> list (S (S m)).
```

In the next example, the local definition is not caught.

```
Coq < Fixpoint length n (l:list n) {struct l} : nat :=
Coq <   match l with
Coq <   | nil => 0
Coq <   | cons n l0 => S (length (2 * n) l0)
Coq <   end.
length is recursively defined (decreasing on 2nd argument)
```

But in this example, it is.

```
Coq < Fixpoint length' n (l:list n) {struct l} : nat :=
Coq <   match l with
Coq <   | nil => 0
Coq <   | cons _ m l0 => S (length' m l0)
Coq <   end.
length' is recursively defined (decreasing on 2nd argument)
```

Remark: for a given matching clause, either none of the local definitions or all of them can be caught.

16.6 Pattern-matching and coercions

If a mismatch occurs between the expected type of a pattern and its actual type, a coercion made from constructors is sought. If such a coercion can be found, it is automatically inserted around the pattern.

Example:

```
Coq < Inductive I : Set :=
Coq <   | C1 : nat -> I
Coq <   | C2 : I -> I.
I is defined
I_rect is defined
I_ind is defined
I_rec is defined

Coq < Coercion C1 : nat >-> I.
C1 is now a coercion

Coq < Check (fun x => match x with
Coq <                 | C2 O => 0
Coq <                 | _ => 0
Coq <                 end).
fun x : I =>
match x with
| C1 _ => 0
| C2 (C1 0) => 0
| C2 (C1 (S _)) => 0
| C2 (C2 _) => 0
end
     : I -> nat
```

16.7 When does the expansion strategy fail ?

The strategy works very like in ML languages when treating patterns of non-dependent type. But there are new cases of failure that are due to the presence of dependencies.

The error messages of the current implementation may be sometimes confusing. When the tactic fails because patterns are somehow incorrect then error messages refer to the initial expression. But the strategy may succeed to build an expression whose sub-expressions are well typed when the whole expression is not. In this situation the message makes reference to the expanded expression. We encourage users, when they have patterns with the same outer constructor in different equations, to name the variable patterns in the same positions with the same name. E.g. to write `(cons n O x) => e1` and `(cons n _ x) => e2` instead of `(cons n O x) => e1` and `(cons n' _ x') => e2`. This helps to maintain certain name correspondence between the generated expression and the original.

Here is a summary of the error messages corresponding to each situation:

Error messages:

1. `The constructor` *ident* `expects` *num* `arguments`

   `The variable` *ident* `is bound several times in pattern` *term*

   `Found a constructor of inductive type` *term* `while a constructor of` *term* `is expected`

Patterns are incorrect (because constructors are not applied to the correct number of the arguments, because they are not linear or they are wrongly typed).

2. `Non exhaustive pattern-matching`

   The pattern matching is not exhaustive.

3. `The elimination predicate` *term* `should be of arity` *num* `(for non dependent case) or` *num* `(for dependent case)`

   The elimination predicate provided to `match` has not the expected arity.

4. `Unable to infer a match predicate`
   `Either there is a type incompatiblity or the problem involves dependencies`

   There is a type mismatch between the different branches. The user should provide an elimination predicate.

Chapter 17

Implicit Coercions

Amokrane Saïbi

17.1 General Presentation

This section describes the inheritance mechanism of Coq. In Coq with inheritance, we are not interested in adding any expressive power to our theory, but only convenience. Given a term, possibly not typable, we are interested in the problem of determining if it can be well typed modulo insertion of appropriate coercions. We allow to write:

- $f\ a$ where $f : forall\ x : A, B$ and $a : A'$ when A' can be seen in some sense as a subtype of A.
- $x : A$ when A is not a type, but can be seen in a certain sense as a type: set, group, category etc.
- $f\ a$ when f is not a function, but can be seen in a certain sense as a function: bijection, functor, any structure morphism etc.

17.2 Classes

A class with n parameters is any defined name with a type $forall\ (x_1 : A_1)..(x_n : A_n), s$ where s is a sort. Thus a class with parameters is considered as a single class and not as a family of classes. An object of a class C is any term of type $C\ t_1..t_n$. In addition to these user-classes, we have two abstract classes:

- `Sortclass`, the class of sorts; its objects are the terms whose type is a sort.
- `Funclass`, the class of functions; its objects are all the terms with a functional type, i.e. of form $forall\ x : A, B$.

Formally, the syntax of a classes is defined on Figure 17.1.

class	::=	*qualid*
	|	`Sortclass`
	|	`Funclass`

Figure 17.1: Syntax of classes

17.3 Coercions

A name f can be declared as a coercion between a source user-class C with n parameters and a target class D if one of these conditions holds:

- D is a user-class, then the type of f must have the form $forall\ (x_1 : A_1)..(x_n : A_n)(y : C\ x_1..x_n), D\ u_1..u_m$ where m is the number of parameters of D.
- D is `Funclass`, then the type of f must have the form $forall\ (x_1 : A_1)..(x_n : A_n)(y : C\ x_1..x_n)(x : A), B$.
- D is `Sortclass`, then the type of f must have the form $forall\ (x_1 : A_1)..(x_n : A_n)(y : C\ x_1..x_n), s$ with s a sort.

We then write $f : C>->D$. The restriction on the type of coercions is called *the uniform inheritance condition*. Remark that the abstract classes `Funclass` and `Sortclass` cannot be source classes.

To coerce an object $t : C\ t_1..t_n$ of C towards D, we have to apply the coercion f to it; the obtained term $f\ t_1..t_n\ t$ is then an object of D.

17.4 Identity Coercions

Identity coercions are special cases of coercions used to go around the uniform inheritance condition. Let C and D be two classes with respectively n and m parameters and $f : forall\ (x_1 : T_1)..(x_k : T_k)(y : C\ u_1..u_n), D\ v_1..v_m$ a function which does not verify the uniform inheritance condition. To declare f as coercion, one has first to declare a subclass C' of C:

$$C' := fun\ (x_1 : T_1)..(x_k : T_k) => C\ u_1..u_n$$

We then define an *identity coercion* between C' and C:

$$Id\_C'\_C \quad := \quad fun\ (x_1 : T_1)..(x_k : T_k)(y : C'\ x_1..x_k) => (y : C\ u_1..u_n)$$

We can now declare f as coercion from C' to D, since we can "cast" its type as $forall\ (x_1 : T_1)..(x_k : T_k)(y : C'\ x_1..x_k), D\ v_1..v_m$.
The identity coercions have a special status: to coerce an object $t : C'\ t_1..t_k$ of C' towards C, we does not have to insert explicitly $Id\_C'\_C$ since $Id\_C'\_C\ t_1..t_k\ t$ is convertible with t. However we "rewrite" the type of t to become an object of C; in this case, it becomes $C\ u_1^*..u_k^*$ where each u_i^* is the result of the substitution in u_i of the variables x_j by t_j.

17.5 Inheritance Graph

Coercions form an inheritance graph with classes as nodes. We call *coercion path* an ordered list of coercions between two nodes of the graph. A class C is said to be a subclass of D if there is a coercion path in the graph from C to D; we also say that C inherits from D. Our mechanism supports multiple inheritance since a class may inherit from several classes, contrary to simple inheritance where a class inherits from at most one class. However there must be at most one path between two classes. If this is not the case, only the *oldest* one is valid and the others are ignored. So the order of declaration of coercions is important.

We extend notations for coercions to coercion paths. For instance $[f_1; ..; f_k] : C>\!\!->D$ is the coercion path composed by the coercions $f_1..f_k$. The application of a coercion path to a term consists of the successive application of its coercions.

17.6 Declaration of Coercions

17.6.1 `Coercion` *qualid* `:` *class*$_1$ `>->` *class*$_2$ `.`

Declares the construction denoted by *qualid* as a coercion between *class*$_1$ and *class*$_2$.

Error messages:

1. *qualid* `not declared`
2. *qualid* `is already a coercion`
3. `Funclass cannot be a source class`
4. `Sortclass cannot be a source class`
5. *qualid* `is not a function`
6. `Cannot find the source class of` *qualid*
7. `Cannot recognize` *class*$_1$ `as a source class of` *qualid*
8. *qualid* `does not respect the inheritance uniform condition`
9. `Found target class` *class* `instead of` *class*$_2$

When the coercion *qualid* is added to the inheritance graph, non valid coercion paths are ignored; they are signaled by a warning.

Warning :

1. `Ambiguous paths:` $[f_1^1; ..; f_{n_1}^1] : C_1>\!\!->D_1$ ... $[f_1^m; ..; f_{n_m}^m] : C_m>\!\!->D_m$

Variants:

1. `Local Coercion` *qualid* `:` *class*$_1$ `>->` *class*$_2$ `.`
   Declares the construction denoted by *qualid* as a coercion local to the current section.

2. `Coercion` *ident* `:=` *term*
   This defines *ident* just like `Definition` *ident* `:=` *term*, and then declares *ident* as a coercion between it source and its target.

3. `Coercion` *ident* `:=` *term* `:` *type*
   This defines *ident* just like `Definition` *ident* `:` *type* `:=` *term*, and then declares *ident* as a coercion between it source and its target.

4. `Local Coercion` *ident* `:=` *term*
   This defines *ident* just like `Let` *ident* `:=` *term*, and then declares *ident* as a coercion between it source and its target.

5. Assumptions can be declared as coercions at declaration time. This extends the grammar of declarations from Figure 1.3 as follows:

   declaration ::= *declaration_keyword assums* `.`

   assums ::= *simple_assums*
   | `(` *simple_assums* `)` ... `(` *simple_assums* `)`

   simple_assums ::= *ident* ... *ident* `:` *[>] term*

   If the extra > is present before the type of some assumptions, these assumptions are declared as coercions.

6. Constructors of inductive types can be declared as coercions at definition time of the inductive type. This extends and modifies the grammar of inductive types from Figure 1.3 as follows:

   inductive ::= `Inductive` *ind_body* `with` ... `with` *ind_body* `.`
   | `CoInductive` *ind_body* `with` ... `with` *ind_body* `.`

   ind_body ::= *ident [binderlet* ... *binderlet] : term :=*
   [[|] constructor | ... | constructor]

   constructor ::= *ident [binderlet* ... *binderlet] [: [>] term]*

   Especially, if the extra > is present in a constructor declaration, this constructor is declared as a coercion.

17.6.2 `Identity Coercion` *ident* : *class*$_1$ `>->` *class*$_2$ `.`

We check that *class*$_1$ is a constant with a value of the form $fun\ (x_1 : T_1)..(x_n : T_n) => (\mathit{class}_2\ t_1..t_m)$ where m is the number of parameters of *class*$_2$. Then we define an identity function with the type $forall\ (x_1 : T_1)..(x_n : T_n)(y : \mathit{class}_1\ x_1..x_n), \mathit{class}_2\ t_1..t_m$, and we declare it as an identity coercion between *class*$_1$ and *class*$_2$.

Error messages:

1. *class*$_1$ `must be a transparent constant`

Variants:

1. `Local Identity Coercion` *ident* : *ident*$_1$ `>->` *ident*$_2$ `.`
   Idem but locally to the current section.

2. `SubClass` *ident* `:=` *type* `.`
   If *type* is a class *ident'* applied to some arguments then *ident* is defined and an identity coercion of name `Id_`*ident*`_`*ident'* is declared. Otherwise said, this is an abbreviation for

   `Definition` *ident* `:=` *type* `.`

   followed by

   `Identity Coercion Id_`*ident*`_`*ident'* : *ident* `>->` *ident'* `.`

3. `Local SubClass` *ident* `:=` *type* `.`
   Same as before but locally to the current section.

17.7 Displaying Available Coercions

17.7.1 `Print Classes.`

Print the list of declared classes in the current context.

17.7.2 `Print Coercions.`

Print the list of declared coercions in the current context.

17.7.3 `Print Graph.`

Print the list of valid coercion paths in the current context.

17.7.4 `Print Coercion Paths` *class*$_1$ *class*$_2$ `.`

Print the list of valid coercion paths from *class*$_1$ to *class*$_2$.

17.8 Activating the Printing of Coercions

17.8.1 `Set Printing Coercions.`

This command forces all the coercions to be printed. Conversely, to skip the printing of coercions, use `Unset Printing Coercions`. By default, coercions are not printed.

17.8.2 `Set Printing Coercion` *qualid* `.`

This command forces coercion denoted by *qualid* to be printed. To skip the printing of coercion *qualid*, use `Unset Printing Coercion` *qualid*. By default, a coercion is never printed.

17.9 Classes as Records

We allow the definition of *Structures with Inheritance* (or classes as records) by extending the existing `Record` macro (see Section 2.1). Its new syntax is:

```
Record [>] ident binderlet : sort := [ident0] {
  ident1 [:|:>] term1 ;
  ...
  identn [:|:>] termn }.
```

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. The alternative [:|:>] is ":" or ":>". If $ident_i$:> $term_i$, then $ident_i$ is automatically declared as coercion from *ident* to the class of $term_i$. Remark that $ident_i$ always verifies the uniform inheritance condition. If the optional ">" before *ident* is present, then $ident_0$ (or the default name `Build_`*ident* if $ident_0$ is omitted) is automatically declared as a coercion from the class of $term_n$ to *ident* (this may fail if the uniform inheritance condition is not satisfied).

Remark: The keyword `Structure` is a synonym of `Record`.

17.10 Coercions and Sections

The inheritance mechanism is compatible with the section mechanism. The global classes and coercions defined inside a section are redefined after its closing, using their new value and new type. The classes and coercions which are local to the section are simply forgotten. Coercions with a local source class or a local target class, and coercions which do not verify the uniform inheritance condition any longer are also forgotten.

17.11 Examples

There are three situations:

- $f\ a$ is ill-typed where $f : forall\ x : A, B$ and $a : A'$. If there is a coercion path between A' and A, $f\ a$ is transformed into $f\ a'$ where a' is the result of the application of this coercion path to a.

  We first give an example of coercion between atomic inductive types

```
Coq < Definition bool_in_nat (b:bool) := if b then 0 else 1.
bool_in_nat is defined
Coq < Coercion bool_in_nat : bool >-> nat.
bool_in_nat is now a coercion
Coq < Check (0 = true).
0 = true
     : Prop
Coq < Set Printing Coercions.
Coq < Check (0 = true).
0 = bool_in_nat true
     : Prop
```

Warning: "`Check true=0.`" fails. This is "normal" behaviour of coercions. To validate `true=0`, the coercion is searched from `nat` to `bool`. There is none.

We give an example of coercion between classes with parameters.

```
Coq < Parameters
Coq <    (C : nat -> Set) (D : nat -> bool -> Set) (E : bool -> Set).
C is assumed
D is assumed
E is assumed

Coq < Parameter f : forall n:nat, C n -> D (S n) true.
f is assumed

Coq < Coercion f : C >-> D.
f is now a coercion

Coq < Parameter g : forall (n:nat) (b:bool), D n b -> E b.
g is assumed

Coq < Coercion g : D >-> E.
g is now a coercion

Coq < Parameter c : C 0.
c is assumed

Coq < Parameter T : E true -> nat.
T is assumed

Coq < Check (T c).
T c
     : nat

Coq < Set Printing Coercions.

Coq < Check (T c).
T (g 1 true (f 0 c))
     : nat
```

We give now an example using identity coercions.

```
Coq < Definition D' (b:bool) := D 1 b.
D' is defined

Coq < Identity Coercion IdD'D : D' >-> D.

Coq < Print IdD'D.
IdD'D =
(fun (b : bool) (x : D' b) => x):forall b : bool, D' b -> D 1 b
     : forall b : bool, D' b -> D 1 b
Argument scopes are [bool_scope _]

Coq < Parameter d' : D' true.
d' is assumed

Coq < Check (T d').
T d'
     : nat

Coq < Set Printing Coercions.
```

```
Coq < Check (T d').
T (g 1 true d')
     : nat
```

In the case of functional arguments, we use the monotonic rule of sub-typing. Approximatively, to coerce $t : forall\ x : A, B$ towards $forall\ x : A', B'$, one have to coerce A' towards A and B towards B'. An example is given below:

```
Coq < Parameters (A B : Set) (h : A -> B).
A is assumed
B is assumed
h is assumed

Coq < Coercion h : A >-> B.
h is now a coercion

Coq < Parameter U : (A -> E true) -> nat.
U is assumed

Coq < Parameter t : B -> C 0.
t is assumed

Coq < Check (U t).
U (fun x : A => t x)
     : nat

Coq < Set Printing Coercions.

Coq < Check (U t).
U (fun x : A => g 1 true (f 0 (t (h x))))
     : nat
```

Remark the changes in the result following the modification of the previous example.

```
Coq < Parameter U' : (C 0 -> B) -> nat.
U' is assumed

Coq < Parameter t' : E true -> A.
t' is assumed

Coq < Check (U' t').
U' (fun x : C 0 => t' x)
     : nat

Coq < Set Printing Coercions.

Coq < Check (U' t').
U' (fun x : C 0 => h (t' (g 1 true (f 0 x))))
     : nat
```

- An assumption $x : A$ when A is not a type, is ill-typed. It is replaced by $x : A'$ where A' is the result of the application to A of the coercion path between the class of A and `Sortclass` if it exists. This case occurs in the abstraction $fun\ x : A => t$, universal quantification $forall\ x : A, B$, global variables and parameters of (co-)inductive definitions and functions. In $forall\ x : A, B$, such a coercion path may be applied to B also if necessary.

```
Coq < Parameter Graph : Type.
Graph is assumed

Coq < Parameter Node : Graph -> Type.
Node is assumed

Coq < Coercion Node : Graph >-> Sortclass.
Node is now a coercion

Coq < Parameter G : Graph.
G is assumed

Coq < Parameter Arrows : G -> G -> Type.
Arrows is assumed

Coq < Check Arrows.
Arrows
     : G -> G -> Type

Coq < Parameter fg : G -> G.
fg is assumed

Coq < Check fg.
fg
     : G -> G

Coq < Set Printing Coercions.

Coq < Check fg.
fg
     : Node G -> Node G
```

- $f\ a$ is ill-typed because $f : A$ is not a function. The term f is replaced by the term obtained by applying to f the coercion path between A and `Funclass` if it exists.

```
Coq < Parameter bij : Set -> Set -> Set.
bij is assumed

Coq < Parameter ap : forall A B:Set, bij A B -> A -> B.
ap is assumed

Coq < Coercion ap : bij >-> Funclass.
ap is now a coercion

Coq < Parameter b : bij nat nat.
b is assumed

Coq < Check (b 0).
b 0
     : nat

Coq < Set Printing Coercions.

Coq < Check (b 0).
ap nat nat b 0
     : nat
```

Let us see the resulting graph of this session.

```
Coq < Print Graph.
[sigT_of_sig; sig_of_sigT] : sig >-> sig
[sigT_of_sig] : sig >-> sigT
[sig_of_sigT] : sigT >-> sig
[sig_of_sigT; sigT_of_sig] : sigT >-> sigT
[bool_in_nat] : bool >-> nat
[f] : C >-> D
[f; g] : C >-> E
[g] : D >-> E
[IdD'D] : D' >-> D
[IdD'D; g] : D' >-> E
[h] : A >-> B
[Node] : Graph >-> Sortclass
[ap] : bij >-> Funclass
```

Chapter 18

Type Classes

Matthieu Sozeau

The status of Type Classes is (extremelly) experimental.

This chapter presents a quick reference of the commands related to type classes. For an actual introduction to type classes, there is a description of the system [126] and the literature on type classes in HASKELL which also applies.

18.1 Class and Instance declarations

The syntax for class and instance declarations is the same as record syntax of COQ:

$$\texttt{Class Id}\ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n)[\text{: } \mathit{sort}] := \{$$
$$\quad \mathsf{f}_1 \ : \ \mathit{type}_1;$$
$$\quad \vdots$$
$$\quad \mathsf{f}_m \ : \ \mathit{type}_m\}.$$

$$\texttt{Instance}\ \mathit{ident}\ \texttt{: Id}\ \mathit{term}_1 \cdots \mathit{term}_n := \{$$
$$\quad \mathsf{f}_1 \ := \ \mathit{term}_{f_1};$$
$$\quad \vdots$$
$$\quad \mathsf{f}_m \ := \ \mathit{term}_{f_m}\}.$$

The $\overrightarrow{\alpha_i : \tau_i}$ variables are called the *parameters* of the class and the $\overrightarrow{f_k : \mathit{type}_k}$ are called the *methods*. Each class definition gives rise to a corresponding record declaration and each instance is a regular definition whose name is given by *ident* and type is an instantiation of the record type.

We'll use the following example class in the rest of the chapter:

```
Coq < Class EqDec (A : Type) := {
Coq <   eqb : A -> A -> bool ;
Coq <   eqb_leibniz : forall x y, eqb x y = true -> x = y }.
```

This class implements a boolean equality test which is compatible with leibniz equality on some type. An example implementation is:

```
Coq < Instance unit_EqDec : EqDec unit :=
Coq < { eqb x y := true ;
Coq <   eqb_leibniz x y H :=
Coq <     match x, y return x = y with tt, tt => refl_equal tt end }.
```

If one does not give all the members in the `Instance` declaration, Coq enters the proof-mode and the user is asked to build inhabitants of the remaining fields, e.g.:

```
Coq < Instance eq_bool : EqDec bool :=
Coq < { eqb x y := if x then y else negb y }.

Coq < Proof. intros x y H.
1 subgoal

  x : bool
  y : bool
  H : (if x then y else negb y) = true
  ============================
   x = y

Coq <   destruct x ; destruct y ; (discriminate || reflexivity).
Proof completed.

Coq < Defined.
refine (Build_EqDec bool (fun x y : bool => if x then y else negb y)
          (_:forall x y : bool, (if x then y else negb y) = true -> x = y)).
intros x y H.
destruct x; destruct y; discriminate || reflexivity.
eq_bool is defined
```

One has to take care that the transparency of every field is determined by the transparency of the `Instance` proof. One can use alternatively the `Program Instance` variant which has richer facilities for dealing with obligations.

18.2 Binding classes

Once a type class is declared, one can use it in class binders:

```
Coq < Definition neqb {A} {eqa : EqDec A} (x y : A) := negb (eqb x y).
neqb is defined
```

When one calls a class method, a constraint is generated that is satisfied only in contexts where the appropriate instances can be found. In the example above, a constraint `EqDec A` is generated and satisfied by `eqa : EqDec A`. In case no satisfying constraint can be found, an error is raised:

```
Coq < Definition neqb' (A : Type) (x y : A) := negb (eqb x y).
Toplevel input, characters 47-50:
> Definition neqb' (A : Type) (x y : A) := negb (eqb x y).
>                                               ^^^
Error: Cannot infer the implicit parameter EqDec of
eqb.
```

```
Could not find an instance for "EqDec ?11" in environment:
A : Type
x : A
y : A
```

The algorithm used to solve constraints is a variant of the eauto tactic that does proof search with a set of lemmas (the instances). It will use local hypotheses as well as declared lemmas in the `typeclass_instances` database. Hence the example can also be written:

```
Coq < Definition neqb' A (eqa : EqDec A) (x y : A) := negb (eqb x y).
neqb' is defined
```

However, the generalizing binders should be used instead as they have particular support for type classes:

- They automatically set the maximally implicit status for type class arguments, making derived functions as easy to use as class methods. In the example above, `A` and `eqa` should be set maximally implicit.
- They support implicit quantification on class arguments and partialy applied type classes (§18.2.1)
- They support implicit quantification on superclasses (§18.4.1)

18.2.1 Implicit quantification

Implicit quantification is an automatic elaboration of a statement with free variables into a closed statement where these variables are quantified explicitely. Implicit generalization is done only inside binders begining with a backquote ` and the codomain of `Instance` declarations.

Following the previous example, one can write:

```
Coq < Definition neqb_impl `{eqa : EqDec A} (x y : A) := negb (eqb x y).
neqb_impl is defined
```

Here `A` is implicitely generalized, and the resulting function is equivalent to the one above. One must be careful that *all* the free variables are generalized, which may result in confusing errors in case of typos. In such cases, the context will probably contain some unexpected generalized variable.

The generalizing binders `` `{ } `` and `` `( ) `` work similarly to their explicit counterparts, only binding the generalized variables implicitly, as maximally-inserted arguments. In these binders, the binding name for the bound object is optional, whereas the type is mandatory, dually to regular binders.

18.3 Parameterized Instances

One can declare parameterized instances as in HASKELL simply by giving the constraints as a binding context before the instance, e.g.:

```
Coq < Instance prod_eqb `(EA : EqDec A, EB : EqDec B) : EqDec (A * B) :=
Coq < { eqb x y := match x, y with
Coq <   | (la, ra), (lb, rb) => andb (eqb la lb) (eqb ra rb)
Coq <   end }.
1 subgoal
```

```
A : Type
EA : EqDec A
B : Type
EB : EqDec B
============================
 forall x y : A * B,
 (let (la, ra) := x in let (lb, rb) := y in (eqb la lb && eqb ra rb)%bool) =
 true -> x = y
```

These instances are used just as well as lemmas in the instance hint database.

18.4 Building hierarchies

18.4.1 Superclasses

One can also parameterize classes by other classes, generating a hierarchy of classes and superclasses. In the same way, we give the superclasses as a binding context:

```
Coq < Class Ord `(E : EqDec A) :=
Coq <   { le : A -> A -> bool }.
```

Contrary to HASKELL, we have no special syntax for superclasses, but this declaration is morally equivalent to:

```
Class `(E : EqDec A) => Ord A :=
  { le : A -> A -> bool }.
```

This declaration means that any instance of the `Ord` class must have an instance of `EqDec`. The parameters of the subclass contain at least all the parameters of its superclasses in their order of appearance (here `A` is the only one). As we have seen, `Ord` is encoded as a record type with two parameters: a type `A` and an `E` of type `EqDec A`. However, one can still use it as if it had a single parameter inside generalizing binders: the generalization of superclasses will be done automatically.

```
Coq < Definition le_eqb `{Ord A} (x y : A) := andb (le x y) (le y x).
```

In some cases, to be able to specify sharing of structures, one may want to give explicitely the superclasses. It is is possible to do it directly in regular binders, and using the `!` modifier in class binders. For example:

```
Coq < Definition lt `{eqa : EqDec A, ! Ord eqa} (x y : A) :=
Coq <   andb (le x y) (neqb x y).
```

The `!` modifier switches the way a binder is parsed back to the regular interpretation of Coq. In particular, it uses the implicit arguments mechanism if available, as shown in the example.

18.4.2 Substructures

Substructures are components of a class which are instances of a class themselves. They often arise when using classes for logical properties, e.g.:

```
Coq < Class Reflexive (A : Type) (R : relation A) :=
Coq <   reflexivity : forall x, R x x.

Coq < Class Transitive (A : Type) (R : relation A) :=
Coq <   transitivity : forall x y z, R x y -> R y z -> R x z.
```

This declares singleton classes for reflexive and transitive relations, (see 1 for an explanation). These may be used as part of other classes:

```
Coq < Class PreOrder (A : Type) (R : relation A) :=
Coq < { PreOrder_Reflexive :> Reflexive A R ;
Coq <   PreOrder_Transitive :> Transitive A R }.
```

The syntax `:>` indicates that each `PreOrder` can be seen as a `Reflexive` relation. So each time a reflexive relation is needed, a preorder can be used instead. This is very similar to the coercion mechanism of `Structure` declarations. The implementation simply declares each projection as an instance.

One can also declare existing objects or structure projections using the `Existing Instance` command to achieve the same effect.

18.5 Summary of the commands

18.5.1 `Class` *ident* *binder*$_1$ ... *binder*$_n$: *sort* := { field$_1$; ...; field$_k$ }.

The `Class` command is used to declare a type class with parameters *binder*$_1$ to *binder*$_n$ and fields field$_1$ to field$_k$.

Variants:

1. `Class` *ident* *binder*$_1$ ... *binder*$_n$: *sort* := *ident*$_1$: *type*$_1$. This variant declares a *singleton* class whose only method is *ident*$_1$. This singleton class is a so-called definitional class, represented simply as a definition *ident* *binder*$_1$ ... *binder*$_n$:= *type*$_1$ and whose instances are themselves objects of this type. Definitional classes are not wrapped inside records, and the trivial projection of an instance of such a class is convertible to the instance itself. This can be useful to make instances of existing objects easily and to reduce proof size by not inserting useless projections. The class constant itself is declared rigid during resolution so that the class abstraction is maintained.

18.5.2 `Instance` *ident* *binder*$_1$ ... *binder*$_n$: `Class` t$_1$ ... t$_n$ [| *priority*] := { field$_1$:= b$_1$; ...; field$_i$:= b$_i$ }

The `Instance` command is used to declare a type class instance named *ident* of the class *Class* with parameters t$_1$ to t$_n$ and fields b$_1$ to b$_i$, where each field must be a declared field of the class. Missing fields must be filled in interactive proof mode.

An arbitrary context of the form *binder*$_1$ ... *binder*$_n$ can be put after the name of the instance and before the colon to declare a parameterized instance. An optional *priority* can be declared, 0 being the highest priority as for auto hints.

Variants:

1. `Instance` *ident* *binder*$_1$ ... *binder*$_n$: `Class` t$_1$ ... t$_n$ [| *priority*] := *term* This syntax is used for declaration of singleton class instances. It does not include curly braces and one need not even mention the unique field name.

2. `Global Instance` One can use the `Global` modifier on instances declared in a section so that their generalization is automatically redeclared after the section is closed.

3. `Program Instance` Switches the type-checking to PROGRAM (chapter 22) and uses the obligation mechanism to manage missing fields.

Besides the `Class` and `Instance` vernacular commands, there are a few other commands related to type classes.

18.5.3 `Existing Instance` *ident*

This commands adds an arbitrary constant whose type ends with an applied type class to the instance database. It can be used for redeclaring instances at the end of sections, or declaring structure projections as instances. This is almost equivalent to `Hint Resolve` *ident* `: typeclass_instances`.

18.5.4 `Typeclasses Transparent, Opaque` $ident_1 \ldots ident_n$

This commands defines the transparency of $ident_1 \ldots ident_n$ during type class resolution. It is useful when some constants prevent some unifications and make resolution fail. It is also useful to declare constants which should never be unfolded during proof-search, like fixpoints or anything which does not look like an abbreviation. This can additionally speed up proof search as the typeclass map can be indexed by such rigid constants (see 8.13.1). By default, all constants and local variables are considered transparent. One should take care not to make opaque any constant that is used to abbreviate a type, like `relation A := A -> A -> Prop`.

This is equivalent to `Hint Transparent,Opaque` *ident* `: typeclass_instances`.

18.5.5 `Typeclasses eauto := [debug] [dfs | bfs] [`*depth*`]`

This commands allows to customize the type class resolution tactic, based on a variant of eauto. The flags semantics are:

- `debug` In debug mode, the trace of successfully applied tactics is printed.
- `dfs, bfs` This sets the search strategy to depth-first search (the default) or breadth-first search.
- *depth* This sets the depth of the search (the default is 100).

Chapter 19

Omega: a solver of quantifier-free problems in Presburger Arithmetic

Pierre Crégut

19.1 Description of omega

omega solves a goal in Presburger arithmetic, i.e. a universally quantified formula made of equations and inequations. Equations may be specified either on the type nat of natural numbers or on the type Z of binary-encoded integer numbers. Formulas on nat are automatically injected into Z. The procedure may use any hypothesis of the current proof session to solve the goal.

Multiplication is handled by omega but only goals where at least one of the two multiplicands of products is a constant are solvable. This is the restriction meant by "Presburger arithmetic".

If the tactic cannot solve the goal, it fails with an error message. In any case, the computation eventually stops.

19.1.1 Arithmetical goals recognized by omega

omega applied only to quantifier-free formulas built from the connectors

```
/\, \/, ~, ->
```

on atomic formulas. Atomic formulas are built from the predicates

```
=, le, lt, gt, ge
```

on nat or from the predicates

```
=, <, <=, >, >=
```

on Z. In expressions of type nat, omega recognizes

```
plus, minus, mult, pred, S, O
```

and in expressions of type Z, omega recognizes

+, −, *, Zsucc, and constants.

All expressions of type nat or Z not built on these operators are considered abstractly as if they were arbitrary variables of type nat or Z.

19.1.2 Messages from omega

When omega does not solve the goal, one of the following errors is generated:

Error messages:

1. omega can't solve this system

   This may happen if your goal is not quantifier-free (if it is universally quantified, try intros first; if it contains existentials quantifiers too, omega is not strong enough to solve your goal). This may happen also if your goal contains arithmetical operators unknown from omega. Finally, your goal may be really wrong!

2. omega: Not a quantifier-free goal

   If your goal is universally quantified, you should first apply intro as many time as needed.

3. omega: Unrecognized predicate or connective: *ident*

4. omega: Unrecognized atomic proposition: *prop*

5. omega: Can't solve a goal with proposition variables

6. omega: Unrecognized proposition

7. omega: Can't solve a goal with non-linear products

8. omega: Can't solve a goal with equality on *type*

19.2 Using omega

The omega tactic does not belong to the core system. It should be loaded by

```
Coq < Require Import Omega.
Coq < Open Scope Z_scope.
```

Example 3:

```
Coq < Goal forall m n:Z, 1 + 2 * m <> 2 * n.
1 subgoal

  ============================
   forall m n : Z, 1 + 2 * m <> 2 * n

Coq < intros; omega.
Proof completed.
```

Example 4:

```
Coq < Goal forall z:Z, z > 0 -> 2 * z + 1 > z.
1 subgoal

  ============================
   forall z : Z, z > 0 -> 2 * z + 1 > z

Coq < intro; omega.
Proof completed.
```

19.3 Technical data

19.3.1 Overview of the tactic

- The goal is negated twice and the first negation is introduced as an hypothesis.
- Hypothesis are decomposed in simple equations or inequations. Multiple goals may result from this phase.
- Equations and inequations over `nat` are translated over `Z`, multiple goals may result from the translation of substraction.
- Equations and inequations are normalized.
- Goals are solved by the *OMEGA* decision procedure.
- The script of the solution is replayed.

19.3.2 Overview of the *OMEGA* decision procedure

The *OMEGA* decision procedure involved in the `omega` tactic uses a small subset of the decision procedure presented in

> "The Omega Test: a fast and practical integer programming algorithm for dependence analysis", William Pugh, Communication of the ACM , 1992, p 102-114.

Here is an overview, look at the original paper for more information.

- Equations and inequations are normalized by division by the GCD of their coefficients.
- Equations are eliminated, using the Banerjee test to get a coefficient equal to one.
- Note that each inequation defines a half space in the space of real value of the variables.
- Inequations are solved by projecting on the hyperspace defined by cancelling one of the variable. They are partitioned according to the sign of the coefficient of the eliminated variable. Pairs of inequations from different classes define a new edge in the projection.
- Redundant inequations are eliminated or merged in new equations that can be eliminated by the Banerjee test.

- The last two steps are iterated until a contradiction is reached (success) or there is no more variable to eliminate (failure).

It may happen that there is a real solution and no integer one. The last steps of the Omega procedure (dark shadow) are not implemented, so the decision procedure is only partial.

19.4 Bugs

- The simplification procedure is very dumb and this results in many redundant cases to explore.
- Much too slow.
- Certainly other bugs! You can report them to

Pierre.Cregut@cnet.francetelecom.fr

Chapter 20

Micromega : tactics for solving arithmetics goals over ordered rings

Frédéric Besson and Evgeny Makarov

For using the tactics out-of-the-box, read Section 20.1. Section 20.2 presents some background explaining the proof principle for solving polynomials goals. Section 20.3 explains how to get a complete procedure for linear integer arithmetic.

20.1 The `psatz` tactic in a hurry

Load the `Psatz` module (`Require Psatz.`). This module defines the tactics: `lia`, `psatzl D`, and `psatz D n` where `D` is `Z`, `Q` or `R` and `n` is an optional integer limiting the proof search depth.

- The `psatzl` tactic solves linear goals using an embedded (naive) linear programming prover *i.e.*, fourier elimination.
- The `psatz` tactic solves polynomial goals using John Harrison's Hol light driver to the external prover `cspd`[1]. Note that the `csdp` driver is generating a *proof cache* thus allowing to rerun scripts even without `csdp`.
- The `lia` (linear integer arithmetic) tactic is specialised to solve linear goals over $\mathbb{Z}$. It extends `psatzl Z` and exploits the discreetness of $\mathbb{Z}$.

These tactics solve propositional formulas parameterised by atomic arithmetics expressions interpreted over a domain $D \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$. The syntax of the formulas is the following:

$$\begin{array}{lcl} F & ::= & A \mid P \mid \mathit{True} \mid \mathit{False} \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \leftrightarrow F_2 \mid F_1 \rightarrow F_2 \mid \sim F \\ A & ::= & p_1 = p_2 \mid p_1 > p_2 \mid p_1 < p_2 \mid p_1 \geq p_2 \mid p_1 \leq p_2 \\ p & ::= & c \mid x \mid -p \mid p_1 - p_2 \mid p_1 + p_2 \mid p_1 \times p_2 \mid p\hat{}n \end{array}$$

[1] Sources and binaries can be found at https://projects.coin-or.org/Csdp

where c is a numeric constant, $x \in D$ is a numeric variable and the operators $-$, $+$, $\times$, are respectively subtraction, addition, product, $p\hat{}n$ is exponentiation by a constant n, P is an arbitrary proposition. The following table details for each domain $D \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$ the range of constants c and exponent n.

	$\mathbb{Z}$	$\mathbb{Q}$	$\mathbb{R}$
c	`Z`	`Q`	$\{R1, R0\}$
n	`Z`	`Z`	`nat`

20.2 *Positivstellensatz* refutations

The name `psatz` is an abbreviation for *positivstellensatz* – literally positivity theorem – which generalises Hilbert's *nullstellensatz*. It relies on the notion of $Cone$. Given a (finite) set of polynomials S, $Cone(S)$ is inductively defined as the smallest set of polynomials closed under the following rules:

$$\frac{p \in S}{p \in Cone(S)} \qquad \frac{}{p^2 \in Cone(S)} \qquad \frac{p_1 \in Cone(S) \quad p_2 \in Cone(S) \quad \bowtie \in \{+, *\}}{p_1 \bowtie p_2 \in Cone(S)}$$

The following theorem provides a proof principle for checking that a set of polynomial inequalities do not have solutions[2]:

Theorem 1 *Let S be a set of polynomials.*
If -1 belongs to $Cone(S)$ then the conjunction $\bigwedge_{p \in S} p \geq 0$ is unsatisfiable.

A proof based on this theorem is called a *positivstellensatz* refutation. The tactics work as follows. Formulas are normalised into conjonctive normal form $\bigwedge_i C_i$ where C_i has the general form $(\bigwedge_{j \in S_i} p_j \bowtie 0) \rightarrow \mathit{False})$ and $\bowtie \in \{>, \geq, =\}$ for $D \in \{\mathbb{Q}, \mathbb{R}\}$ and $\bowtie \in \{\geq, =\}$ for $\mathbb{Z}$. For each conjunct C_i, the tactic calls a oracle which searches for -1 within the cone. Upon success, the oracle returns a *cone expression* that is normalised by the `ring` tactic (see chapter 23) and checked to be -1.

To illustrate the working of the tactic, consider we wish to prove the following Coq goal.

```
Coq <   Goal forall x, -x^2 >= 0 -> x - 1 >= 0 -> False.
```

Such a goal is solved by `intro x; psatz Z`. The oracle returns the cone expression $2 \times (\mathbf{x - 1}) + \mathbf{x - 1} \times \mathbf{x - 1} + -\mathbf{x^2}$ (polynomial hypotheses are printed in bold). By construction, this expression belongs to $Cone(\{-x^2, x - 1\})$. Moreover, by running `ring` we obtain -1. By Theorem 1, the goal is valid.

The `psatzl` tactic is searching for *linear* refutations using a fourier elimination[3]. As a result, this tactic explore a subset of the $Cone$ defined as:

$$LinCone(S) = \left\{ \sum_{p \in S} \alpha_p \times p \;\middle|\; \alpha_p \text{ are positive constants} \right\}$$

Basically, the deductive power of `psatzl` is the combined deductive power of `ring_simplify` and `fourier`.

[2] Variants deal with equalities and strict inequalities.
[3] More efficient linear programming techniques could equally be employed

The psatz tactic explores the $Cone$ by increasing degrees – hence the depth parameter n. In theory, such a proof search is complete – if the goal is provable the search eventually stops. Unfortunately, the external oracle is using numeric (approximate) optimisation techniques that might miss a refutation.

20.3 lia : the linear integer arithmetic tactic

The tactic lia offers an alternative to the omega and romega tactic (see Chapter 19). It solves goals that omega and romega do not solve, such as the following so-called *omega nightmare* [120].

```
Coq <   Goal forall x y,
Coq <        27 <= 11 * x + 13 * y <= 45 ->
Coq <        -10 <= 7 * x - 9 * y <= 4 ->   False.
```

The estimation of the relative efficiency of lia *vs* omega and romega is under evaluation.

High level view of lia. Over $\mathbb{R}$, *positivstellensatz* refutations are a complete proof principle[4]. However, this is not the case over $\mathbb{Z}$. Actually, *positivstellensatz* refutations are not even sufficient to decide linear *integer* arithmetics. The canonical exemple is `2 * x = 1 -> False` which is a theorem of $\mathbb{Z}$ but not a theorem of $\mathbb{R}$. To remedy this weakness, the lia tactic is using recursively a combination of:

- linear *positivstellensatz* refutations *i.e.*, psatzl Z;
- cutting plane proofs;
- case split.

Cutting plane proofs are a way to take into account the discreetness of $\mathbb{Z}$ by rounding up (rational) constants up-to the closest integer.

Theorem 2 *Let p be an integer and c a rational constant.*

$$p \geq c \Rightarrow p \geq \lceil c \rceil$$

For instance, from $2 * x = 1$ we can deduce

- $x \geq 1/2$ which cut plane is $x \geq \lceil 1/2 \rceil = 1$;
- $x \leq 1/2$ which cut plane is $x \leq \lfloor 1/2 \rfloor = 0$.

By combining these two facts (in normal form) $x - 1 \geq 0$ and $-x \geq 0$, we conclude by exhibiting a *positivstellensatz* refutation ($-1 \equiv \mathbf{x} - \mathbf{1} + -\mathbf{x} \in Cone(\{x - 1, x\})$).

Cutting plane proofs and linear *positivstellensatz* refutations are a complete proof principle for integer linear arithmetic.

Case split allow to enumerate over the possible values of an expression.

Theorem 3 *Let p be an integer and c_1 and c_2 integer constants.*

$$c_1 \leq p \leq c_2 \Rightarrow \bigvee_{x \in [c_1, c_2]} p = x$$

Our current oracle tries to find an expression e with a small range $[c_1, c_2]$. We generate $c_2 - c_1$ subgoals which contexts are enriched with an equation $e = i$ for $i \in [c_1, c_2]$ and recursively search for a proof.

[4] In practice, the oracle might fail to produce such a refutation.

Chapter 21

Extraction of programs in Objective Caml and Haskell

Jean-Christophe Filliâtre and Pierre Letouzey

The status of extraction is experimental.

We present here the COQ extraction commands, used to build certified and relatively efficient functional programs, extracting them from the proofs of their specifications. The functional languages available as output are currently OBJECTIVE CAML, HASKELL and SCHEME. In the following, "ML" will be used (abusively) to refer to any of the three.

Differences with old versions. The current extraction mechanism is new for version 7.0 of COQ. In particular, the F_ω toplevel used as an intermediate step between COQ and ML has been withdrawn. It is also not possible any more to import ML objects in this F_ω toplevel. The current mechanism also differs from the one in previous versions of COQ: there is no more an explicit toplevel for the language (formerly called FML).

21.1 Generating ML code

The next two commands are meant to be used for rapid preview of extraction. They both display extracted term(s) inside COQ.

`Extraction` *qualid* **.**

Extracts one constant or module in the COQ toplevel.

`Recursive Extraction` $qualid_1$ `...` $qualid_n$ **.**

Recursive extraction of all the globals (or modules) $qualid_1 \ldots qualid_n$ and all their dependencies in the COQ toplevel.

All the following commands produce real ML files. User can choose to produce one monolithic file or one file per COQ library.

`Extraction "`*file*`"` *qualid*$_1$ ... *qualid*$_n$.

Recursive extraction of all the globals (or modules) *qualid*$_1$ ... *qualid*$_n$ and all their dependencies in one monolithic file *file*. Global and local identifiers are renamed according to the choosen ML language to fullfill its syntactic conventions, keeping original names as much as possible.

`Extraction Library` *ident*.

Extraction of the whole COQ library *ident*`.v` to an ML module *ident*`.ml`. In case of name clash, identifiers are here renamed using prefixes `coq_` or `Coq_` to ensure a session-independent renaming.

`Recursive Extraction Library` *ident*.

Extraction of the COQ library *ident*`.v` and all other modules *ident*`.v` depends on.

The list of globals *qualid*$_i$ does not need to be exhaustive: it is automatically completed into a complete and minimal environment.

21.2 Extraction options

21.2.1 Setting the target language

The ability to fix target language is the first and more important of the extraction options. Default is Ocaml. Besides Haskell and Scheme, another language called Toplevel is provided. It is a pseudo-Ocaml, with no renaming on global names: so names are printed as in COQ. This third language is available only at the COQ Toplevel.

```
Extraction Language Ocaml.

Extraction Language Haskell.

Extraction Language Scheme.

Extraction Language Toplevel.
```

21.2.2 Inlining and optimizations

Since Objective Caml is a strict language, the extracted code has to be optimized in order to be efficient (for instance, when using induction principles we do not want to compute all the recursive calls but only the needed ones). So the extraction mechanism provides an automatic optimization routine that will be called each time the user want to generate Ocaml programs. Essentially, it performs constants inlining and reductions. Therefore some constants may not appear in resulting monolithic Ocaml program (a warning is printed for each such constant). In the case of modular extraction, even if some inlining is done, the inlined constant are nevertheless printed, to ensure session-independent programs.

Concerning Haskell, such optimizations are less useful because of lazyness. We still make some optimizations, for example in order to produce more readable code.

All these optimizations are controled by the following COQ options:

```
Set Extraction Optimize.
```

`Unset Extraction Optimize.`

Default is Set. This control all optimizations made on the ML terms (mostly reduction of dummy beta/iota redexes, but also simplications on Cases, etc). Put this option to Unset if you want a ML term as close as possible to the Coq term.

`Set Extraction AutoInline.`

`Unset Extraction AutoInline.`

Default is Set, so by default, the extraction mechanism feels free to inline the bodies of some defined constants, according to some heuristics like size of bodies, useness of some arguments, etc. Those heuristics are not always perfect, you may want to disable this feature, do it by Unset.

`Extraction Inline` *qualid*$_1$ ... *qualid*$_n$.

`Extraction NoInline` *qualid*$_1$ ... *qualid*$_n$.

In addition to the automatic inline feature, you can now tell precisely to inline some more constants by the `Extraction Inline` command. Conversely, you can forbid the automatic inlining of some specific constants by the `Extraction NoInline` command. Those two commands enable a precise control of what is inlined and what is not.

`Print Extraction Inline.`

Prints the current state of the table recording the custom inlinings declared by the two previous commands.

`Reset Extraction Inline.`

Puts the table recording the custom inlinings back to empty.

Inlining and printing of a constant declaration. A user can explicitely asks a constant to be extracted by two means:

- by mentioning it on the extraction command line
- by extracting the whole COQ module of this constant.

In both cases, the declaration of this constant will be present in the produced file. But this same constant may or may not be inlined in the following terms, depending on the automatic/custom inlining mechanism.

For the constants non-explicitely required but needed for dependancy reasons, there are two cases:

- If an inlining decision is taken, wether automatically or not, all occurences of this constant are replaced by its extracted body, and this constant is not declared in the generated file.
- If no inlining decision is taken, the constant is normally declared in the produced file.

21.2.3 Realizing axioms

Extraction will fail if it encounters an informative axiom not realized (see Section 21.2.3). A warning will be issued if it encounters an logical axiom, to remind user that inconsistant logical axioms may lead to incorrect or non-terminating extracted terms.

It is possible to assume some axioms while developing a proof. Since these axioms can be any kind of proposition or object or type, they may perfectly well have some computational content. But a program must be a closed term, and of course the system cannot guess the program which realizes an axiom. Therefore, it is possible to tell the system what ML term corresponds to a given axiom.

`Extract Constant` *qualid* `=>` *string* `.`

Give an ML extraction for the given constant. The *string* may be an identifier or a quoted string.

`Extract Inlined Constant` *qualid* `=>` *string* `.`

Same as the previous one, except that the given ML terms will be inlined everywhere instead of being declared via a let.

Note that the `Extract Inlined Constant` command is sugar for an `Extract Constant` followed by a `Extraction Inline`. Hence a `Reset Extraction Inline` will have an effect on the realized and inlined xaxiom.

Of course, it is the responsability of the user to ensure that the ML terms given to realize the axioms do have the expected types. In fact, the strings containing realizing code are just copied in the extracted files. The extraction recognize whether the realized axiom should become a ML type constant or a ML object declaration.

Example:

```
Coq < Axiom X:Set.
X is assumed
Coq < Axiom x:X.
x is assumed
Coq < Extract Constant X => "int".
Coq < Extract Constant x => "0".
```

Notice that in the case of type scheme axiom (i.e. whose type is an arity, that is a sequence of product finished by a sort), then some type variables has to be given. The syntax is then:

`Extract Constant` *qualid* *string*$_1$ `...` *string*$_n$ `=>` *string* `.`

The number of type variables is checked by the system.

Example:

```
Coq < Axiom Y : Set -> Set -> Set.
Y is assumed
Coq < Extract Constant Y "'a" "'b" => " 'a*'b ".
```

Realizing an axiom via `Extract Constant` is only useful in the case of an informative axiom (of sort Type or Set). A logical axiom have no computational content and hence will not appears in extracted terms. But a warning is nonetheless issued if extraction encounters a logical axiom. This

warning reminds user that inconsistant logical axioms may lead to incorrect or non-terminating extracted terms.

If an informative axiom has not been realized before an extraction, a warning is also issued and the definition of the axiom is filled with an exception labelled `AXIOM TO BE REALIZED`. The user must then search these exceptions inside the extracted file and replace them by real code.

The system also provides a mechanism to specify ML terms for inductive types and constructors. For instance, the user may want to use the ML native boolean type instead of Coq one. The syntax is the following:

`Extract Inductive` *qualid* `=>` *string* `[` *string* `...` *string* `].`

Give an ML extraction for the given inductive type. You must specify extractions for the type itself (first *string*) and all its constructors (between square brackets). The ML extraction must be an ML recursive datatype.

Example: Typical examples are the following:

```
Coq < Extract Inductive unit => "unit" [ "()" ].
Coq < Extract Inductive bool => "bool" [ "true" "false" ].
Coq < Extract Inductive sumbool => "bool" [ "true" "false" ].
```

If an inductive constructor or type has arity 2 and the corresponding string is enclosed by parenthesis, then the rest of the string is used as infix constructor or type.

```
Coq < Extract Inductive list => "list" [ "[]" "(::)" ].
Toplevel input, characters 18-22:
> Extract Inductive list => "list" [ "[]" "(::)" ].
>                   ^^^^
Error: The reference list was not found in the current environment.
Coq < Extract Inductive prod => "(*)"  [ "(,)" ].
```

21.2.4 Avoiding conflicts with existing filenames

When using `Extraction Library`, the names of the extracted files directly depends from the names of the Coq files. It may happen that these filenames are in conflict with already existing files, either in the standard library of the target language or in other code that is meant to be linked with the extracted code. For instance the module `List` exists both in Coq and in Ocaml. It is possible to instruct the extraction not to use particular filenames.

`Extraction Blacklist` *ident* `...` *ident* `.`

Instruct the extraction to avoid using these names as filenames for extracted code.

`Print Extraction Blacklist.`

Show the current list of filenames the extraction should avoid.

`Reset Extraction Blacklist.`

Allow the extraction to use any filename.

For Ocaml, a typical use of these commands is `Extraction Blacklist String List.`

21.3 Differences between Coq and ML type systems

Due to differences between Coq and ML type systems, some extracted programs are not directly typable in ML. We now solve this problem (at least in Ocaml) by adding when needed some unsafe casting `Obj.magic`, which give a generic type `'a` to any term.

For example, Here are two kinds of problem that can occur:

- If some part of the program is *very* polymorphic, there may be no ML type for it. In that case the extraction to ML works all right but the generated code may be refused by the ML type-checker. A very well known example is the *distr-pair* function:

  ```
  Definition dp :=
   fun (A B:Set)(x:A)(y:B)(f:forall C:Set, C->C) => (f A x, f B y).
  ```

  In Ocaml, for instance, the direct extracted term would be:

  ```
  let dp x y f = Pair((f () x),(f () y))
  ```

  and would have type:

  ```
  dp : 'a -> 'a -> (unit -> 'a -> 'b) -> ('b,'b) prod
  ```

  which is not its original type, but a restriction.

  We now produce the following correct version:

  ```
  let dp x y f = Pair ((Obj.magic f () x), (Obj.magic f () y))
  ```

- Some definitions of Coq may have no counterpart in ML. This happens when there is a quantification over types inside the type of a constructor; for example:

  ```
  Inductive anything : Set := dummy : forall A:Set, A -> anything.
  ```

  which corresponds to the definition of an ML dynamic type. In Ocaml, we must cast any argument of the constructor dummy.

Even with those unsafe castings, you should never get error like "segmentation fault". In fact even if your program may seem ill-typed to the Ocaml type-checker, it can't go wrong: it comes from a Coq well-typed terms, so for example inductives will always have the correct number of arguments, etc.

More details about the correctness of the extracted programs can be found in [91].

We have to say, though, that in most "realistic" programs, these problems do not occur. For example all the programs of Coq library are accepted by Caml type-checker without any `Obj.magic` (see examples below).

21.4 Some examples

We present here two examples of extractions, taken from the Coq Standard Library. We choose Objective Caml as target language, but all can be done in the other dialects with slight modifications. We then indicate where to find other examples and tests of Extraction.

21.4.1 A detailed example: Euclidean division

The file `Euclid` contains the proof of Euclidean division (theorem `eucl_dev`). The natural numbers defined in the example files are unary integers defined by two constructors O and S:

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

This module contains a theorem `eucl_dev`, and its extracted term is of type

```
forall b:nat, b > 0 -> forall a:nat, diveucl a b
```

where `diveucl` is a type for the pair of the quotient and the modulo. We can now extract this program to OBJECTIVE CAML:

```
Coq < Require Import Euclid.
Coq < Extraction Inline Wf_nat.gt_wf_rec Wf_nat.lt_wf_rec.
Coq < Recursive Extraction  eucl_dev.
type __ = Obj.t
let __ = let rec f _ = Obj.repr f in Obj.repr f
type nat =
  | O
  | S of nat
type sumbool =
  | Left
  | Right
(** val minus : nat -> nat -> nat **)
let rec minus n m =
  match n with
    | O -> n
    | S k -> (match m with
                | O -> n
                | S l -> minus k l)
(** val le_lt_dec : nat -> nat -> sumbool **)
let rec le_lt_dec n m =
  match n with
    | O -> Left
    | S n0 -> (match m with
                 | O -> Right
                 | S m0 -> le_lt_dec n0 m0)
(** val le_gt_dec : nat -> nat -> sumbool **)
let le_gt_dec n m =
  le_lt_dec n m
(** val induction_ltof2 :
    ('a1 -> nat) -> ('a1 -> ('a1 -> __ -> 'a2) -> 'a2) -> 'a1 -> 'a2 **)
let rec induction_ltof2 f x a =
  x a (fun y _ -> induction_ltof2 f x y)
type diveucl =
  | Divex of nat * nat
(** val eucl_dev : nat -> nat -> diveucl **)
let eucl_dev b a =
  induction_ltof2 (fun m -> m) (fun n h0 ->
```

```
    match le_gt_dec b n with
      | Left -> let Divex (x, x0) = h0 (minus n b) __ in Divex ((S x), x0)
      | Right -> Divex (O, n)) a
```

The inlining of `gt_wf_rec` and `lt_wf_rec` is not mandatory. It only enhances readability of extracted code. You can then copy-paste the output to a file `euclid.ml` or let COQ do it for you with the following command:

```
Coq < Extraction "euclid" eucl_dev.
The file euclid.ml has been created by extraction.
The file euclid.mli has been created by extraction.
```

Let us play the resulting program:

```
# #use "euclid.ml";;
type sumbool = Left | Right
type nat = O | S of nat
type diveucl = Divex of nat * nat
val minus : nat -> nat -> nat = <fun>
val le_lt_dec : nat -> nat -> sumbool = <fun>
val le_gt_dec : nat -> nat -> sumbool = <fun>
val eucl_dev : nat -> nat -> diveucl = <fun>
# eucl_dev (S (S O)) (S (S (S (S (S O)))));;
- : diveucl = Divex (S (S O), S O)
```

It is easier to test on OBJECTIVE CAML integers:

```
# let rec i2n = function 0 -> O | n -> S (i2n (n-1));;
val i2n : int -> nat = <fun>
# let rec n2i = function O -> 0 | S p -> 1+(n2i p);;
val n2i : nat -> int = <fun>
# let div a b =
     let Divex (q,r) = eucl_dev (i2n b) (i2n a) in (n2i q, n2i r);;
div : int -> int -> int * int = <fun>
# div 173 15;;
- : int * int = 11, 8
```

21.4.2 Another detailed example: Heapsort

The file `Heap.v` contains the proof of an efficient list sorting algorithm described by Bjerner. Is is an adaptation of the well-known *heapsort* algorithm to functional languages. The main function is `treesort`, whose type is shown below:

```
Coq < Require Import Heap.
Coq < Check treesort.
treesort
     : forall (A : Type) (leA eqA : relation A),
       (forall x y : A, {leA x y} + {leA y x}) ->
       forall eqA_dec : forall x y : A, {eqA x y} + {~ eqA x y},
       (forall x y z : A, leA x y -> leA y z -> leA x z) ->
       forall l : list A,
       {m : list A | sort leA m &  permutation eqA eqA_dec l m}
```

Let's now extract this function:

```
Coq < Extraction Inline sort_rec is_heap_rec.

Coq < Extraction NoInline list_to_heap.

Coq < Extraction "heapsort" treesort.
The file heapsort.ml has been created by extraction.
The file heapsort.mli has been created by extraction.
```

One more time, the `Extraction Inline` and `NoInline` directives are cosmetic. Without it, everything goes right, but the output is less readable. Here is the produced file `heapsort.ml`:

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

type 'a sig2 =
  'a
  (* singleton inductive, whose constructor was exist2 *)

type sumbool =
  | Left
  | Right

type 'a list =
  | Nil
  | Cons of 'a * 'a list

type 'a multiset =
  'a -> nat
  (* singleton inductive, whose constructor was Bag *)

type 'a merge_lem =
  'a list
  (* singleton inductive, whose constructor was merge_exist *)

(** val merge : ('a1 -> 'a1 -> sumbool) -> ('a1 -> 'a1 -> sumbool) ->
                'a1 list -> 'a1 list -> 'a1 merge_lem **)

let rec merge leA_dec eqA_dec l1 l2 =
  match l1 with
    | Nil -> l2
    | Cons (a, l) ->
        let rec f = function
          | Nil -> Cons (a, l)
          | Cons (a0, l3) ->
              (match leA_dec a a0 with
                 | Left -> Cons (a,
```

```
                    (merge leA_dec eqA_dec l (Cons (a0, l3))))
                | Right -> Cons (a0, (f l3)))
        in f l2

type 'a tree =
  | Tree_Leaf
  | Tree_Node of 'a * 'a tree * 'a tree

type 'a insert_spec =
  'a tree
  (* singleton inductive, whose constructor was insert_exist *)

(** val insert : ('a1 -> 'a1 -> sumbool) -> ('a1 -> 'a1 -> sumbool) ->
                 'a1 tree -> 'a1 -> 'a1 insert_spec **)

let rec insert leA_dec eqA_dec t a =
  match t with
    | Tree_Leaf -> Tree_Node (a, Tree_Leaf, Tree_Leaf)
    | Tree_Node (a0, t0, t1) ->
        let h3 = fun x -> insert leA_dec eqA_dec t0 x in
        (match leA_dec a0 a with
           | Left -> Tree_Node (a0, t1, (h3 a))
           | Right -> Tree_Node (a, t1, (h3 a0)))

type 'a build_heap =
  'a tree
  (* singleton inductive, whose constructor was heap_exist *)

(** val list_to_heap : ('a1 -> 'a1 -> sumbool) -> ('a1 -> 'a1 ->
                       sumbool) -> 'a1 list -> 'a1 build_heap **)

let rec list_to_heap leA_dec eqA_dec = function
  | Nil -> Tree_Leaf
  | Cons (a, l0) ->
      insert leA_dec eqA_dec (list_to_heap leA_dec eqA_dec l0) a

type 'a flat_spec =
  'a list
  (* singleton inductive, whose constructor was flat_exist *)

(** val heap_to_list : ('a1 -> 'a1 -> sumbool) -> ('a1 -> 'a1 ->
                       sumbool) -> 'a1 tree -> 'a1 flat_spec **)

let rec heap_to_list leA_dec eqA_dec = function
  | Tree_Leaf -> Nil
  | Tree_Node (a, t0, t1) -> Cons (a,
      (merge leA_dec eqA_dec (heap_to_list leA_dec eqA_dec t0)
```

```
        (heap_to_list leA_dec eqA_dec t1)))

(** val treesort : ('a1 -> 'a1 -> sumbool) -> ('a1 -> 'a1 -> sumbool)
                   -> 'a1 list -> 'a1 list sig2 **)

let treesort leA_dec eqA_dec l =
  heap_to_list leA_dec eqA_dec (list_to_heap leA_dec eqA_dec l)
```

Let's test it:

```
# #use "heapsort.ml";;
type sumbool = Left | Right
type nat = O | S of nat
type 'a tree = Tree_Leaf | Tree_Node of 'a * 'a tree * 'a tree
type 'a list = Nil | Cons of 'a * 'a list
val merge :
  ('a -> 'a -> sumbool) -> 'b -> 'a list -> 'a list -> 'a list = <fun>
val heap_to_list :
  ('a -> 'a -> sumbool) -> 'b -> 'a tree -> 'a list = <fun>
val insert :
  ('a -> 'a -> sumbool) -> 'b -> 'a tree -> 'a -> 'a tree = <fun>
val list_to_heap :
  ('a -> 'a -> sumbool) -> 'b -> 'a list -> 'a tree = <fun>
val treesort :
  ('a -> 'a -> sumbool) -> 'b -> 'a list -> 'a list = <fun>
```

One can remark that the argument of `treesort` corresponding to `eqAdec` is never used in the informative part of the terms, only in the logical parts. So the extracted `treesort` never use it, hence this `'b` argument. We will use `()` for this argument. Only remains the `leAdec` argument (of type `'a -> 'a -> sumbool`) to really provide.

```
# let leAdec x y = if x <= y then Left else Right;;
val leAdec : 'a -> 'a -> sumbool = <fun>
# let rec listn = function 0 -> Nil
                         | n -> Cons(Random.int 10000,listn (n-1));;
val listn : int -> int list = <fun>
# treesort leAdec () (listn 9);;
- : int list = Cons (160, Cons (883, Cons (1874, Cons (3275, Cons
  (5392, Cons (7320, Cons (8512, Cons (9632, Cons (9876, Nil)))))))))
```

Some tests on longer lists (10000 elements) show that the program is quite efficient for Caml code.

21.4.3 The Standard Library

As a test, we propose an automatic extraction of the Standard Library of Coq. In particular, we will find back the two previous examples, `Euclid` and `Heapsort`. Go to directory `contrib/extraction/test` of the sources of Coq, and run commands:

```
make tree; make
```

This will extract all Standard Library files and compile them. It is done via many `Extraction Module`, with some customization (see subdirectory `custom`).

This test works also with Haskell. In the same directory, run:

```
make tree; make -f Makefile.haskell
```

The haskell compiler currently used is `hbc`. Any other should also work, just adapt the `Makefile.haskell`. In particular `ghc` is known to work.

21.4.4 Extraction's horror museum

Some pathological examples of extraction are grouped in the file

`contrib/extraction/test_extraction.v`

of the sources of Coq.

21.4.5 Users' Contributions

Several of the Coq Users' Contributions use extraction to produce certified programs. In particular the following ones have an automatic extraction test (just run `make` in those directories):

- Bordeaux/Additions
- Bordeaux/EXCEPTIONS
- Bordeaux/SearchTrees
- Dyade/BDDS
- Lannion
- Lyon/CIRCUITS
- Lyon/FIRING-SQUAD
- Marseille/CIRCUITS
- Muenchen/Higman
- Nancy/FOUnify
- Rocq/ARITH/Chinese
- Rocq/COC
- Rocq/GRAPHS
- Rocq/HIGMAN
- Sophia-Antipolis/Stalmarck
- Suresnes/BDD

Lannion, Rocq/HIGMAN and Lyon/CIRCUITS are a bit particular. They are the only examples of developments where `Obj.magic` are needed. This is probably due to an heavy use of impredicativity. After compilation those two examples run nonetheless, thanks to the correction of the extraction [91].

Chapter 22

PROGRAM

Matthieu Sozeau

The status of PROGRAM *is experimental.*

We present here the new PROGRAM tactic commands, used to build certified COQ programs, elaborating them from their algorithmic skeleton and a rich specification [125]. It can be sought of as a dual of extraction (see Chapter 21). The goal of PROGRAM is to program as in a regular functional programming language whilst using as rich a specification as desired and proving that the code meets the specification using the whole COQ proof apparatus. This is done using a technique originating from the "Predicate subtyping" mechanism of PVS[122], which generates type-checking conditions while typing a term constrained to a particular type. Here we insert existential variables in the term, which must be filled with proofs to get a complete COQ term. PROGRAM replaces the PROGRAM tactic by Catherine Parent [111] which had a similar goal but is no longer maintained.

The languages available as input are currently restricted to COQ's term language, but may be extended to OBJECTIVE CAML, HASKELL and others in the future. We use the same syntax as COQ and permit to use implicit arguments and the existing coercion mechanism. Input terms and types are typed in an extended system (RUSSELL) and interpreted into COQ terms. The interpretation process may produce some proof obligations which need to be resolved to create the final term.

22.1 Elaborating programs

The main difference from COQ is that an object in a type $T : \mathsf{Set}$ can be considered as an object of type $\{x : T \mid P\}$ for any wellformed $P : \mathsf{Prop}$. If we go from T to the subset of T verifying property P, we must prove that the object under consideration verifies it. RUSSELL will generate an obligation for every such coercion. In the other direction, RUSSELL will automatically insert a projection.

Another distinction is the treatment of pattern-matching. Apart from the following differences, it is equivalent to the standard `match` operation (see Section 4.5.4).

- Generation of equalities. A `match` expression is always generalized by the corresponding equality. As an example, the expression:

```
Coq <   match x with
Coq <   | 0 => t
Coq <   | S n => u
Coq <   end.
```

will be first rewrote to:

```
Coq <   (match x as y return (x = y -> _) with
Coq <   | 0 => fun H : x = 0 -> t
Coq <   | S n => fun H : x = S n -> u
Coq <   end) (refl_equal n).
```

This permits to get the proper equalities in the context of proof obligations inside clauses, without which reasoning is very limited.

- Generation of inequalities. If a pattern intersects with a previous one, an inequality is added in the context of the second branch. See for example the definition of `div2` below, where the second branch is typed in a context where $\forall p, \_ <> S(Sp)$.
- Coercion. If the object being matched is coercible to an inductive type, the corresponding coercion will be automatically inserted. This also works with the previous mechanism.

If you do specify a `return` or `in` clause the typechecker will fall back directly to COQ's usual typing of dependent pattern-matching.

The next two commands are similar to their standard counterparts Definition (see Section 1.3.2) and Fixpoint (see Section 1.3.4) in that they define constants. However, they may require the user to prove some goals to construct the final definitions.

22.1.1 `Program Definition` *ident* `:=` *term* `.`

This command types the value *term* in RUSSELL and generate proof obligations. Once solved using the commands shown below, it binds the final COQ term to the name *ident* in the environment.

Error messages:

1. *ident* `already exists`

Variants:

1. `Program Definition` *ident* `:` $term_1$ `:=` $term_2$ `.`
   It interprets the type $term_1$, potentially generating proof obligations to be resolved. Once done with them, we have a COQ type $term'_1$. It then checks that the type of the interpretation of $term_2$ is coercible to $term'_1$, and registers *ident* as being of type $term'_1$ once the set of obligations generated during the interpretation of $term_2$ and the aforementioned coercion derivation are solved.
2. `Program Definition` *ident* $binder_1 \ldots binder_n$ `:` $term_1$ `:=` $term_2$ `.`
   This is equivalent to
   `Program Definition` *ident* `: forall` $binder_1 \ldots binder_n$`,` $term_1$ `:=`
   `fun` $binder_1 \ldots binder_n$ `=>` $term_2$ `.`

Error messages:

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

See also: Sections 6.9.1, 6.9.2, 8.5.5

22.1.2 `Program Fixpoint` *ident params* `{order} :` `type :=` *term*

The structural fixpoint operator behaves just like the one of Coq (see Section 1.3.4), except it may also generate obligations. It works with mutually recursive definitions too.

```
Coq < Require Import Program.

Coq < Program Fixpoint div2 (n : nat) : { x : nat | n = 2 * x \/ n = 2 * x + 1 } :=
Coq <   match n with
Coq <   | S (S p) => S (div2 p)
Coq <   | _ => O
Coq <   end.
Solving obligations automatically...
4 obligations remaining
```

Here we have one obligation for each branch (branches for `0` and `(S 0)` are automatically generated by the pattern-matching compilation algorithm).

```
Coq <   Obligation 1.
1 subgoal

  div2 : forall n : nat,
         {x : nat | n = x + (x + 0) \/ n = x + (x + 0) + 1}
  p : nat
  ============================
   S (S p) = S ('(div2 p) + S ('(div2 p) + 0)) \/
   S (S p) = S ('(div2 p) + S ('(div2 p) + 0) + 1)
```

One can use a well-founded order or a measure as termination orders using the syntax:

```
Coq < Definition id (n : nat) := n.

Coq < Program Fixpoint div2 (n : nat) {measure id n} :
Coq <   { x : nat | n = 2 * x \/ n = 2 * x + 1 } :=
Coq <   match n with
Coq <   | S (S p) => S (div2 p)
Coq <   | _ => O
Coq <   end.
```

The `measure` keyword expects a measure function into the naturals, whereas `wf` expects a relation.

Caution When defining structurally recursive functions, the generated obligations should have the prototype of the currently defined functional in their context. In this case, the obligations should be transparent (e.g. using Defined) so that the guardedness condition on recursive calls can be checked by the kernel's type-checker. There is an optimization in the generation of obligations which gets rid of the hypothesis corresponding to the functionnal when it is not necessary, so that the obligation can be declared opaque (e.g. using Qed). However, as soon as it appears in the context, the proof of the obligation is *required* to be declared transparent.

No such problems arise when using measures or well-founded recursion.

An important point about well-founded and measure-based functions is the following: The recursive prototype of a function of type $binder_1 \ldots binder_n$ { `measure m` $binder_i$ }:$type_1$, inside the body is $\{binder'_i$ | `m` x'_i < `m` $x_i\} \ldots binder_n$,$type_1$. So any arguments appearing before the recursive one are ignored for the recursive calls, hence they are constant.

22.1.3 Program Lemma *ident* : type.

The RUSSELL language can also be used to type statements of logical properties. It will generate obligations, try to solve them automatically and fail if some unsolved obligations remain. In this case, one can first define the lemma's statement using Program Definition and use it as the goal afterwards. Otherwise the proof will be started with the elobarted version as a goal. The Program prefix can similarly be used as a prefix for Variable, Hypothesis, Axiom etc...

22.2 Solving obligations

The following commands are available to manipulate obligations. The optional identifier is used when multiple functions have unsolved obligations (e.g. when defining mutually recursive blocks). The optional tactic is replaced by the default one if not specified.

- Obligation Tactic := *expr* Sets the default obligation solving tactic applied to all obligations automatically, whether to solve them or when starting to prove one, e.g. using Next.
- Obligations [of *ident*] Displays all remaining obligations.
- Obligation num [of *ident*] Start the proof of obligation num.
- Next Obligation [of *ident*] Start the proof of the next unsolved obligation.
- Solve Obligations [of *ident*] [using *expr*] Tries to solve each obligation of *ident*using the given tactic or the default one.
- Solve All Obligations [using *expr*] Tries to solve each obligation of every program using the given tactic or the default one (useful for mutually recursive definitions).
- Admit Obligations [of *ident*] Admits all obligations (does not work with structurally recursive programs).
- Preterm [of *ident*] Shows the term that will be fed to the kernel once the obligations are solved. Useful for debugging.
- Set Transparent Obligations Control whether all obligations should be declared as transparent (the default), or if the system should infer which obligations can be declared opaque.

The module Coq.Program.Tactics defines the default tactic for solving obligations called program_simpl. Importing Coq.Program.Program also adds some useful notations, as documented in the file itself.

22.3 Frequently Asked Questions

- Ill-formed recursive definitions This error can happen when one tries to define a function by structural recursion on a subset object, which means the Coq function looks like:

  ```
  Program Fixpoint f (x : A | P) := match x with A b => f b end.
  ```

  Supposing $b : A$, the argument at the recursive call to f is not a direct subterm of x as b is wrapped inside an exist constructor to build an object of type {x : A | P}. Hence the definition is rejected by the guardedness condition checker. However you can do wellfounded recursion on subset objects like this:

```
Program Fixpoint f (x : A | P) { measure size } :=
  match x with A b => f b end.
```

You will then just have to prove that the measure decreases at each recursive call. There are three drawbacks though:

1. You have to define the measure yourself;
2. The reduction is a little more involved, although it works using lazy evaluation;
3. Mutual recursion on the underlying inductive type isn't possible anymore, but nested mutual recursion is always possible.

Chapter 23

The `ring` and `field` tactic families

Bruno Barras, Benjamin Grégoire, Assia Mahboubi, Laurent Théry[1]

This chapter presents the tactics dedicated to deal with ring and field equations.

23.1 What does this tactic do?

`ring` does associative-commutative rewriting in ring and semi-ring structures. Assume you have two binary functions $\oplus$ and $\otimes$ that are associative and commutative, with $\oplus$ distributive on $\otimes$, and two constants 0 and 1 that are unities for $\oplus$ and $\otimes$. A *polynomial* is an expression built on variables $V_0, V_1, \ldots$ and constants by application of $\oplus$ and $\otimes$.

Let an *ordered product* be a product of variables $V_{i_1} \otimes \ldots \otimes V_{i_n}$ verifying $i_1 \leq i_2 \leq \cdots \leq i_n$. Let a *monomial* be the product of a constant and an ordered product. We can order the monomials by the lexicographic order on products of variables. Let a *canonical sum* be an ordered sum of monomials that are all different, i.e. each monomial in the sum is strictly less than the following monomial according to the lexicographic order. It is an easy theorem to show that every polynomial is equivalent (modulo the ring properties) to exactly one canonical sum. This canonical sum is called the *normal form* of the polynomial. In fact, the actual representation shares monomials with same prefixes. So what does `ring`? It normalizes polynomials over any ring or semi-ring structure. The basic use of `ring` is to simplify ring expressions, so that the user does not have to deal manually with the theorems of associativity and commutativity.

Examples:

1. In the ring of integers, the normal form of $x(3 + yx + 25(1 - z)) + zx$ is $28x + (-24)xz + xxy$.
2. For the classical propositional calculus (or the boolean rings) the normal form is what logicians call *disjunctive normal form*: every formula is equivalent to a disjunction of conjunctions of atoms. (Here $\oplus$ is $\vee$, $\otimes$ is $\wedge$, variables are atoms and the only constants are T and F)

`ring` is also able to compute a normal form modulo monomial equalities. For example, under the hypothesis that $2x^2 = yz + 1$, the normal form of $2(x + 1)x - x - zy$ is $x + 1$.

[1] based on previous work from Patrick Loiseleur and Samuel Boutin

23.2 The variables map

It is frequent to have an expression built with + and $\times$, but rarely on variables only. Let us associate a number to each subterm of a ring expression in the GALLINA language. For example in the ring `nat`, consider the expression:

```
(plus (mult (plus (f (5)) x) x)
   (mult (if b then (4) else (f (3))) (2)))
```

As a ring expression, it has 3 subterms. Give each subterm a number in an arbitrary order:

0 $\mapsto$ `if b then (4) else (f (3))`
1 $\mapsto$ `(f (5))`
2 $\mapsto$ `x`

Then normalize the "abstract" polynomial

$$((V_1 \otimes V_2) \oplus V_2) \oplus (V_0 \otimes 2)$$

In our example the normal form is:

$$(2 \otimes V_0) \oplus (V_1 \otimes V_2) \oplus (V_2 \otimes V_2)$$

Then substitute the variables by their values in the variables map to get the concrete normal polynomial:

```
(plus (mult (2) (if b then (4) else (f (3))))
   (plus (mult (f (5)) x) (mult x x)))
```

23.3 Is it automatic?

Yes, building the variables map and doing the substitution after normalizing is automatically done by the tactic. So you can just forget this paragraph and use the tactic according to your intuition.

23.4 Concrete usage in COQ

The `ring` tactic solves equations upon polynomial expressions of a ring (or semi-ring) structure. It proceeds by normalizing both hand sides of the equation (w.r.t. associativity, commutativity and distributivity, constant propagation, rewriting of monomials) and comparing syntactically the results.

`ring_simplify` applies the normalization procedure described above to the terms given. The tactic then replaces all occurrences of the terms given in the conclusion of the goal by their normal forms. If no term is given, then the conclusion should be an equation and both hand sides are normalized. The tactic can also be applied in a hypothesis.

The tactic must be loaded by `Require Import Ring`. The ring structures must be declared with the `Add Ring` command (see below). The ring of booleans is predefined; if one wants to use the tactic on `nat` one must first require the module `ArithRing` (exported by `Arith`); for `Z`, do `Require Import ZArithRing` or simply `Require Import ZArith`; for `N`, do `Require Import NArithRing` or `Require Import NArith`.

Example:

```
Coq < Require Import ZArith.
Coq < Open Scope Z_scope.
Coq < Goal forall a b c:Z,
Coq <   (a + b + c)^2  =
Coq <   a * a + b^2 + c * c + 2 * a * b + 2 * a * c + 2 * b * c.
1 subgoal

  ============================
   forall a b c : Z,
   (a + b + c) ^ 2 =
   a * a + b ^ 2 + c * c + 2 * a * b + 2 * a * c + 2 * b * c

Coq < intros; ring.
Proof completed.

Coq < Goal forall a b:Z, 2*a*b = 30 ->
Coq <         (a+b)^2 = a^2 + b^2 + 30.
1 subgoal

  ============================
   forall a b : Z, 2 * a * b = 30 -> (a + b) ^ 2 = a ^ 2 + b ^ 2 + 30

Coq < intros a b H; ring [H].
Proof completed.
```

Variants:

1. `ring [`*term*$_1$ `...` *term*$_n$`]` decides the equality of two terms modulo ring operations and rewriting of the equalities defined by *term*$_1$ ... *term*$_n$. Each of *term*$_1$ ... *term*$_n$ has to be a proof of some equality $m = p$, where m is a monomial (after "abstraction"), p a polynomial and $=$ the corresponding equality of the ring structure.
2. `ring_simplify [`*term*$_1$ `...` *term*$_n$`]` $t_1 \ldots t_m$ `in` *ident* performs the simplification in the hypothesis named `ident`.

Warning: `ring_simplify` *term*$_1$`;` `ring_simplify` *term*$_2$ is not equivalent to `ring_simplify` *term*$_1$ *term*$_2$. In the latter case the variables map is shared between the two terms, and common subterm t of *term*$_1$ and *term*$_2$ will have the same associated variable number. So the first alternative should be avoided for terms belonging to the same ring theory.

Error messages:

1. `not a valid ring equation` The conclusion of the goal is not provable in the corresponding ring theory.
2. `arguments of ring_simplify do not have all the same type` `ring_simplify` cannot simplify terms of several rings at the same time. Invoke the tactic once per ring structure.
3. `cannot find a declared ring structure over term` No ring has been declared for the type of the terms to be simplified. Use `Add Ring` first.
4. `cannot find a declared ring structure for equality term` Same as above is the case of the `ring` tactic.

23.5 Adding a ring structure

Declaring a new ring consists in proving that a ring signature (a carrier set, an equality, and ring operations: `Ring_theory.ring_theory` and `Ring_theory.semi_ring_theory`) satisfies the ring axioms. Semi-rings (rings without + inverse) are also supported. The equality can be either Leibniz equality, or any relation declared as a setoid (see 24.7). The definition of ring and semi-rings (see module `Ring_theory`) is:

```
Record ring_theory : Prop := mk_rt {
   Radd_0_l    : forall x, 0 + x == x;
   Radd_sym    : forall x y, x + y == y + x;
   Radd_assoc  : forall x y z, x + (y + z) == (x + y) + z;
   Rmul_1_l    : forall x, 1 * x == x;
   Rmul_sym    : forall x y, x * y == y * x;
   Rmul_assoc  : forall x y z, x * (y * z) == (x * y) * z;
   Rdistr_l    : forall x y z, (x + y) * z == (x * z) + (y * z);
   Rsub_def    : forall x y, x - y == x + -y;
   Ropp_def    : forall x, x + (- x) == 0
}.

Record semi_ring_theory : Prop := mk_srt {
   SRadd_0_l   : forall n, 0 + n == n;
   SRadd_sym   : forall n m, n + m == m + n ;
   SRadd_assoc : forall n m p, n + (m + p) == (n + m) + p;
   SRmul_1_l   : forall n, 1*n == n;
   SRmul_0_l   : forall n, 0*n == 0;
   SRmul_sym   : forall n m, n*m == m*n;
   SRmul_assoc : forall n m p, n*(m*p) == (n*m)*p;
   SRdistr_l   : forall n m p, (n + m)*p == n*p + m*p
 }.
```

This implementation of `ring` also features a notion of constant that can be parameterized. This can be used to improve the handling of closed expressions when operations are effective. It consists in introducing a type of *coefficients* and an implementation of the ring operations, and a morphism from the coefficient type to the ring carrier type. The morphism needs not be injective, nor surjective. As an example, one can consider the real numbers. The set of coefficients could be the rational numbers, upon which the ring operations can be implemented. The fact that there exists a morphism is defined by the following properties:

```
Record ring_morph : Prop := mkmorph {
   morph0    : [cO] == 0;
   morph1    : [cI] == 1;
   morph_add : forall x y, [x +! y] == [x]+[y];
   morph_sub : forall x y, [x -! y] == [x]-[y];
   morph_mul : forall x y, [x *! y] == [x]*[y];
   morph_opp : forall x, [-!x] == -[x];
   morph_eq  : forall x y, x?=!y = true -> [x] == [y]
 }.
```

```
Record semi_morph : Prop := mkRmorph {
   Smorph0 : [cO] == 0;
   Smorph1 : [cI] == 1;
   Smorph_add : forall x y, [x +! y] == [x]+[y];
   Smorph_mul : forall x y, [x *! y] == [x]*[y];
   Smorph_eq  : forall x y, x?=!y = true -> [x] == [y]
 }.
```

where `c0` and `cI` denote the 0 and 1 of the coefficient set, `+!`, `*!`, `-!` are the implementations of the ring operations, `==` is the equality of the coefficients, `?+!` is an implementation of this equality, and `[x]` is a notation for the image of `x` by the ring morphism.

Since `Z` is an initial ring (and `N` is an initial semi-ring), it can always be considered as a set of coefficients. There are basically three kinds of (semi-)rings:

abstract rings to be used when operations are not effective. The set of coefficients is `Z` (or `N` for semi-rings).

computational rings to be used when operations are effective. The set of coefficients is the ring itself. The user only has to provide an implementation for the equality.

customized ring for other cases. The user has to provide the coefficient set and the morphism.

This implementation of ring can also recognize simple power expressions as ring expressions. A power function is specified by the following property:

```
Section POWER.
 Variable Cpow : Set.
 Variable Cp_phi : N -> Cpow.
 Variable rpow : R -> Cpow -> R.

 Record power_theory : Prop := mkpow_th {
   rpow_pow_N : forall r n, req (rpow r (Cp_phi n)) (pow_N rI rmul r n)
 }.

End POWER.
```

The syntax for adding a new ring is `Add Ring` $name$ `:` $ring$ `(`mod_1`,...,`mod_2`)`. The name is not relevent. It is just used for error messages. The term $ring$ is a proof that the ring signature satisfies the (semi-)ring axioms. The optional list of modifiers is used to tailor the behavior of the tactic. The following list describes their syntax and effects:

abstract declares the ring as abstract. This is the default.

decidable *term* declares the ring as computational. The expression *term* is the correctness proof of an equality test `?=!` (which should be evaluable). Its type should be of the form `forall x y, x?=!y = true → x == y`.

morphism *term* declares the ring as a customized one. The expression *term* is a proof that there exists a morphism between a set of coefficient and the ring carrier (see `Ring_theory.ring_morph` and `Ring_theory.semi_morph`).

setoid *term*$_1$ *term*$_2$ forces the use of given setoid. The expression *term*$_1$ is a proof that the equality is indeed a setoid (see `Setoid.Setoid_Theory`), and *term*$_2$ a proof that the ring operations are morphisms (see `Ring_theory.ring_eq_ext` and `Ring_theory.sring_eq_ext`). This modifier needs not be used if the setoid and morphisms have been declared.

constants [$\mathcal{L}_{tac}$ **]** specifies a tactic expression that, given a term, returns either an object of the coefficient set that is mapped to the expression via the morphism, or returns `InitialRing.NotConstant`. The default behaviour is to map only 0 and 1 to their counterpart in the coefficient set. This is generally not desirable for non trivial computational rings.

preprocess [$\mathcal{L}_{tac}$ **]** specifies a tactic that is applied as a preliminary step for `ring` and `ring_simplify`. It can be used to transform a goal so that it is better recognized. For instance, `S n` can be changed to `plus 1 n`.

postprocess [$\mathcal{L}_{tac}$ **]** specifies a tactic that is applied as a final step for `ring_simplify`. For instance, it can be used to undo modifications of the preprocessor.

power_tac *term* [$\mathcal{L}_{tac}$ **]** allows `ring` and `ring_simplify` to recognize power expressions with a constant positive integer exponent (example: x^2). The term *term* is a proof that a given power function satisfies the specification of a power function (*term* has to be a proof of `Ring_theory.power_theory`) and $\mathcal{L}_{tac}$ specifies a tactic expression that, given a term, "abstracts" it into an object of type `N` whose interpretation via `Cp_phi` (the evaluation function of power coefficient) is the original term, or returns `InitialRing.NotConstant` if not a constant coefficient (i.e. $\mathcal{L}_{tac}$ is the inverse function of `Cp_phi`). See files `contrib/setoid_ring/ZArithRing.v` and `contrib/setoid_ring/RealField.v` for examples. By default the tactic does not recognize power expressions as ring expressions.

sign *term* allows `ring_simplify` to use a minus operation when outputing its normal form, i.e writing $x - y$ instead of $x + (-y)$. The term *term* is a proof that a given sign function indicates expressions that are signed (*term* has to be a proof of `Ring_theory.get_sign`). See `contrib/setoid_ring/IntialRing.v` for examples of sign function.

div *term* allows `ring` and `ring_simplify` to use moniomals with coefficient other than 1 in the rewriting. The term *term* is a proof that a given division function satisfies the specification of an euclidean division function (*term* has to be a proof of `Ring_theory.div_theory`). For example, this function is called when trying to rewrite $7x$ by $2x = z$ to tell that $7 = 3 * 2 + 1$. See `contrib/setoid_ring/IntialRing.v` for examples of div function.

Error messages:

1. `bad ring structure` The proof of the ring structure provided is not of the expected type.

2. `bad lemma for decidability of equality` The equality function provided in the case of a computational ring has not the expected type.

3. `ring` *operation* `should be declared as a morphism` A setoid associated to the carrier of the ring structure as been found, but the ring operation should be declared as morphism. See 24.7.

23.6 How does it work?

The code of `ring` is a good example of tactic written using *reflection*. What is reflection? Basically, it is writing COQ tactics in COQ, rather than in OBJECTIVE CAML. From the philosophical point of view, it is using the ability of the Calculus of Constructions to speak and reason about itself. For the `ring` tactic we used COQ as a programming language and also as a proof environment to build a tactic and to prove it correctness.

The interested reader is strongly advised to have a look at the file `Ring_polynom.v`. Here a type for polynomials is defined:

```
Inductive PExpr : Type :=
  | PEc : C -> PExpr
  | PEX : positive -> PExpr
  | PEadd : PExpr -> PExpr -> PExpr
  | PEsub : PExpr -> PExpr -> PExpr
  | PEmul : PExpr -> PExpr -> PExpr
  | PEopp : PExpr -> PExpr
  | PEpow : PExpr -> N -> PExpr.
```

Polynomials in normal form are defined as:

```
 Inductive Pol : Type :=
  | Pc : C -> Pol
  | Pinj : positive -> Pol -> Pol
  | PX : Pol -> positive -> Pol -> Pol.
```

where `Pinj n P` denotes P in which V_i is replaced by V_{i+n}, and `PX P n Q` denotes $P \otimes V_1^n \oplus Q'$, Q' being Q where V_i is replaced by V_{i+1}.

Variables maps are represented by list of ring elements, and two interpretation functions, one that maps a variables map and a polynomial to an element of the concrete ring, and the second one that does the same for normal forms:

```
Definition PEeval : list R -> PExpr -> R := [...].
Definition Pphi_dev : list R -> Pol -> R := [...].
```

A function to normalize polynomials is defined, and the big theorem is its correctness w.r.t interpretation, that is:

```
Definition norm : PExpr -> Pol := [...].
Lemma Pphi_dev_ok :
   forall l pe npe, norm pe = npe -> PEeval l pe == Pphi_dev l npe.
```

So now, what is the scheme for a normalization proof? Let `p` be the polynomial expression that the user wants to normalize. First a little piece of ML code guesses the type of `p`, the ring theory `T` to use, an abstract polynomial `ap` and a variables map `v` such that `p` is $\beta\delta\iota$-equivalent to `(PEeval v ap)`. Then we replace it by `(Pphi_dev v (norm ap))`, using the main correctness theorem and we reduce it to a concrete expression `p'`, which is the concrete normal form of `p`. This is summarized in this diagram:

$$\begin{array}{rcl} \texttt{p} & \rightarrow_{\beta\delta\iota} & \texttt{(PEeval v ap)} \\ & & =_{\text{(by the main correctness theorem)}} \\ \texttt{p'} & \leftarrow_{\beta\delta\iota} & \texttt{(Pphi\_dev v (norm ap))} \end{array}$$

The user do not see the right part of the diagram. From outside, the tactic behaves like a $\beta\delta\iota$ simplification extended with AC rewriting rules. Basically, the proof is only the application of the main correctness theorem to well-chosen arguments.

23.7 Dealing with fields

The `field` tactic is an extension of the `ring` to deal with rational expresision. Given a rational expression $F = 0$. It first reduces the expression F to a common denominator $N/D = 0$ where N and D are two ring expressions. For example, if we take $F = (1-1/x)x-x+1$, this gives $N = (x-1)x-x^2+x$ and $D = x$. It then calls `ring` to solve $N = 0$. Note that `field` also generates non-zero conditions for all the denominators it encounters in the reduction. In our example, it generates the condition $x \neq 0$. These conditions appear as one subgoal which is a conjunction if there are several denominators. Non-zero conditions are *always* polynomial expressions. For example when reducing the expression $1/(1 + 1/x)$, two side conditions are generated: $x \neq 0$ and $x + 1 \neq 0$. Factorized expressions are broken since a field is an integral domain, and when the equality test on coefficients is complete w.r.t. the equality of the target field, constants can be proven different from zero automatically.

The tactic must be loaded by `Require Import Field`. New field structures can be declared to the system with the `Add Field` command (see below). The field of real numbers is defined in module `RealField` (in textttcontrib/setoid_ring). It is exported by module `Rbase`, so that requiring `Rbase` or `Reals` is enough to use the field tactics on real numbers. Rational numbers in canonical form are also declared as a field in module `Qcanon`.

Example:

```
Coq < Require Import Reals.

Coq < Open Scope R_scope.

Coq < Goal forall x,  x <> 0 ->
Coq <    (1 - 1/x) * x - x + 1 = 0.
1 subgoal

  ============================
   forall x : R, x <> 0 -> (1 - 1 / x) * x - x + 1 = 0

Coq < intros; field; auto.
Proof completed.

Coq < Goal forall x y, y <> 0 -> y = x -> x/y = 1.
1 subgoal

  ============================
   forall x y : R, y <> 0 -> y = x -> x / y = 1

Coq < intros x y H H1; field [H1]; auto.
Proof completed.
```

Variants:

1. `field [`*term*$_1$ `...` *term*$_n$`]` decides the equality of two terms modulo field operations and rewriting of the equalities defined by *term*$_1$ ... *term*$_n$. Each of *term*$_1$ ... *term*$_n$ has to be a proof of some equality $m = p$, where m is a monomial (after "abstraction"), p a polynomial and $=$ the corresponding equality of the field structure. Beware that rewriting works with the equality $m = p$ only if p is a polynomial since rewriting is handled by the underlying `ring` tactic.

2. `field_simplify` performs the simplification in the conclusion of the goal, $F_1 = F_2$ becomes $N_1/D_1 = N_2/D_2$. A normalization step (the same as the one for rings) is then applied to N_1, D_1,

N_2 and D_2. This way, polynomials remain in factorized form during the fraction simplifications. This yields smaller expressions when reducing to the same denominator since common factors can be cancelled.

3. `field_simplify [`*term*$_1$ `...` *term*$_n$`]` performs the simplification in the conclusion of the goal using the equalities defined by *term*$_1$ ... *term*$_n$.

4. `field_simplify [`*term*$_1$ `...` *term*$_n$`]` t_1 `...`t_m performs the simplification in the terms $t_1 \ldots t_m$ of the conclusion of the goal using the equalities defined by *term*$_1$ ... *term*$_n$.

5. `field_simplify in` H performs the simplification in the assumption H.

6. `field_simplify [`*term*$_1$ `...` *term*$_n$`] in` H performs the simplification in the assumption H using the equalities defined by *term*$_1$ ... *term*$_n$.

7. `field_simplify [`*term*$_1$ `...` *term*$_n$`]` t_1 `...`t_m `in` H performs the simplification in the terms $t_1 \ldots t_n$ of the assumption H using the equalities defined by *term*$_1$ ... *term*$_m$.

8. `field_simplify_eq` performs the simplification in the conclusion of the goal removing the denominator. $F_1 = F_2$ becomes $N_1 D_2 = N_2 D_1$.

9. `field_simplify_eq [`*term*$_1$ `...` *term*$_n$`]` performs the simplification in the conclusion of the goal using the equalities defined by *term*$_1$ ... *term*$_n$.

10. `field_simplify_eq` in H performs the simplification in the assumption H.

11. `field_simplify_eq [`*term*$_1$ `...` *term*$_n$`] in` H performs the simplification in the assumption H using the equalities defined by *term*$_1$ ... *term*$_n$.

23.8 Adding a new field structure

Declaring a new field consists in proving that a field signature (a carrier set, an equality, and field operations: `Field_theory.field_theory` and `Field_theory.semi_field_theory`) satisfies the field axioms. Semi-fields (fields without + inverse) are also supported. The equality can be either Leibniz equality, or any relation declared as a setoid (see 24.7). The definition of fields and semi-fields is:

```
Record field_theory : Prop := mk_field {
    F_R : ring_theory rO rI radd rmul rsub ropp req;
    F_1_neq_0 : ~ 1 == 0;
    Fdiv_def : forall p q, p / q == p * / q;
    Finv_l : forall p, ~ p == 0 ->  / p * p == 1
}.

Record semi_field_theory : Prop := mk_sfield {
    SF_SR : semi_ring_theory rO rI radd rmul req;
    SF_1_neq_0 : ~ 1 == 0;
    SFdiv_def : forall p q, p / q == p * / q;
    SFinv_l : forall p, ~ p == 0 ->  / p * p == 1
}.
```

The result of the normalization process is a fraction represented by the following type:

```
Record linear : Type := mk_linear {
   num : PExpr C;
   denum : PExpr C;
   condition : list (PExpr C) }.
```

where `num` and `denum` are the numerator and denominator; `condition` is a list of expressions that have appeared as a denominator during the normalization process. These expressions must be proven different from zero for the correctness of the algorithm.

The syntax for adding a new field is `Add Field` *name* `:` *field* `(`mod_1`,...,`mod_2`)`. The name is not relevent. It is just used for error messages. *field* is a proof that the field signature satisfies the (semi-)field axioms. The optional list of modifiers is used to tailor the behaviour of the tactic. Since field tactics are built upon ring tactics, all mofifiers of the `Add Ring` apply. There is only one specific modifier:

completeness *term* allows the field tactic to prove automatically that the image of non-zero coefficients are mapped to non-zero elements of the field. *term*is a proof of `forall x y, [x] == [y] -> x?=!y = true`, which is the completeness of equality on coefficients w.r.t. the field equality.

23.9 Legacy implementation

Warning: This tactic is the `ring` tactic of previous versions of Coq and it should be considered as deprecated. It will probably be removed in future releases. It has been kept only for compatibility reasons and in order to help moving existing code to the newer implementation described above. For more details, please refer to the Coq Reference Manual, version 8.0.

23.9.1 `legacy ring` *term*$_1$ `...` *term*$_n$

This tactic, written by Samuel Boutin and Patrick Loiseleur, applies associative commutative rewriting on every ring. The tactic must be loaded by `Require Import LegacyRing`. The ring must be declared in the `Add Ring` command. The ring of booleans is predefined; if one wants to use the tactic on `nat` one must first require the module `LegacyArithRing`; for `Z`, do `Require Import LegacyZArithRing`; for `N`, do `Require Import LegacyNArithRing`.

The terms *term*$_1$, ..., *term*$_n$ must be subterms of the goal conclusion. The tactic `ring` normalizes these terms w.r.t. associativity and commutativity and replace them by their normal form.

Variants:

1. `legacy ring` When the goal is an equality $t_1 = t_2$, it acts like `ring_simplify` t_1 t_2 and then solves the equality by reflexivity.

2. `ring_nat` is a tactic macro for `repeat rewrite S_to_plus_one; ring`. The theorem `S_to_plus_one` is a proof that `forall (n:nat), S n = plus (S O) n`.

You can have a look at the files `LegacyRing.v`, `ArithRing.v`, `ZArithRing.v` to see examples of the `Add Ring` command.

23.9.2 Add a ring structure

It can be done in the COQtoplevel (No ML file to edit and to link with COQ). First, `ring` can handle two kinds of structure: rings and semi-rings. Semi-rings are like rings without an opposite to addition. Their precise specification (in GALLINA) can be found in the file

```
contrib/ring/Ring_theory.v
```

The typical example of ring is `Z`, the typical example of semi-ring is `nat`.

The specification of a ring is divided in two parts: first the record of constants ($\oplus$, $\otimes$, 1, 0, $\ominus$) and then the theorems (associativity, commutativity, etc.).

```
Section Theory_of_semi_rings.

Variable A : Type.
Variable Aplus : A -> A -> A.
Variable Amult : A -> A -> A.
Variable Aone : A.
Variable Azero : A.
(* There is also a "weakly decidable" equality on A. That means
  that if (A_eq x y)=true then x=y but x=y can arise when
  (A_eq x y)=false. On an abstract ring the function [x,y:A]false
  is a good choice. The proof of A_eq_prop is in this case easy. *)
Variable Aeq : A -> A -> bool.

Record Semi_Ring_Theory : Prop :=
{ SR_plus_sym  : (n,m:A)[| n + m == m + n |];
  SR_plus_assoc : (n,m,p:A)[| n + (m + p) == (n + m) + p |];

  SR_mult_sym : (n,m:A)[| n*m == m*n |];
  SR_mult_assoc : (n,m,p:A)[| n*(m*p) == (n*m)*p |];
  SR_plus_zero_left :(n:A)[| 0 + n == n|];
  SR_mult_one_left : (n:A)[| 1*n == n |];
  SR_mult_zero_left : (n:A)[| 0*n == 0 |];
  SR_distr_left   : (n,m,p:A) [| (n + m)*p == n*p + m*p |];
  SR_plus_reg_left : (n,m,p:A)[| n + m == n + p |] -> m==p;
  SR_eq_prop : (x,y:A) (Is_true (Aeq x y)) -> x==y
}.

Section Theory_of_rings.

Variable A : Type.

Variable Aplus : A -> A -> A.
Variable Amult : A -> A -> A.
Variable Aone : A.
Variable Azero : A.
Variable Aopp : A -> A.
Variable Aeq : A -> A -> bool.


Record Ring_Theory : Prop :=
{ Th_plus_sym  : (n,m:A)[| n + m == m + n |];
```

```
  Th_plus_assoc : (n,m,p:A)[| n + (m + p) == (n + m) + p |];
  Th_mult_sym : (n,m:A)[| n*m == m*n |];
  Th_mult_assoc : (n,m,p:A)[| n*(m*p) == (n*m)*p |];
  Th_plus_zero_left :(n:A)[| 0 + n == n|];
  Th_mult_one_left : (n:A)[| 1*n == n |];
  Th_opp_def : (n:A) [| n + (-n) == 0 |];
  Th_distr_left   : (n,m,p:A) [| (n + m)*p == n*p + m*p |];
  Th_eq_prop : (x,y:A) (Is_true (Aeq x y)) -> x==y
}.
```

To define a ring structure on A, you must provide an addition, a multiplication, an opposite function and two unities 0 and 1.

You must then prove all theorems that make (A,Aplus,Amult,Aone,Azero,Aeq) a ring structure, and pack them with the `Build_Ring_Theory` constructor.

Finally to register a ring the syntax is:

`Add Legacy Ring` *A Aplus Amult Aone Azero Ainv Aeq T* `[` *c1 ... cn* `].`

where *A* is a term of type `Set`, *Aplus* is a term of type `A->A->A`, *Amult* is a term of type `A->A->A`, *Aone* is a term of type `A`, *Azero* is a term of type `A`, *Ainv* is a term of type `A->A`, *Aeq* is a term of type `A->bool`, *T* is a term of type `(Ring_Theory` *A Aplus Amult Aone Azero Ainv Aeq*`)`. The arguments *c1 ... cn*, are the names of constructors which define closed terms: a subterm will be considered as a constant if it is either one of the terms *c1 ... cn* or the application of one of these terms to closed terms. For `nat`, the given constructors are `S` and `O`, and the closed terms are `O`, `(S O)`, `(S (S O))`, ...

Variants:

1. `Add Legacy Semi Ring` *A Aplus Amult Aone Azero Aeq T* `[` *c1 ... cn* `].`

   There are two differences with the `Add Ring` command: there is no inverse function and the term *T* must be of type `(Semi_Ring_Theory` *A Aplus Amult Aone Azero Aeq*`)`.

2. `Add Legacy Abstract Ring` *A Aplus Amult Aone Azero Ainv Aeq T*`.`

   This command should be used for when the operations of rings are not computable; for example the real numbers of `theories/REALS/`. Here $0 + 1$ is not beta-reduced to 1 but you still may want to *rewrite* it to 1 using the ring axioms. The argument `Aeq` is not used; a good choice for that function is `[x:A]false`.

3. `Add Legacy Abstract Semi Ring` *A Aplus Amult Aone Azero Aeq T*`.`

Error messages:

1. `Not a valid (semi)ring theory.`

   That happens when the typing condition does not hold.

Currently, the hypothesis is made than no more than one ring structure may be declared for a given type in `Set` or `Type`. This allows automatic detection of the theory used to achieve the normalization. On popular demand, we can change that and allow several ring structures on the same set.

The table of ring theories is compatible with the Coq sectioning mechanism. If you declare a ring inside a section, the declaration will be thrown away when closing the section. And when you load a compiled file, all the `Add Ring` commands of this file that are not inside a section will be loaded.

The typical example of ring is Z, and the typical example of semi-ring is nat. Another ring structure is defined on the booleans.

Warning: Only the ring of booleans is loaded by default with the Ring module. To load the ring structure for nat, load the module ArithRing, and for Z, load the module ZArithRing.

23.9.3 legacy field

This tactic written by David Delahaye and Micaela Mayero solves equalities using commutative field theory. Denominators have to be non equal to zero and, as this is not decidable in general, this tactic may generate side conditions requiring some expressions to be non equal to zero. This tactic must be loaded by Require Import LegacyField. Field theories are declared (as for legacy ring) with the Add Legacy Field command.

23.9.4 Add Legacy Field

This vernacular command adds a commutative field theory to the database for the tactic field. You must provide this theory as follows:

Add Legacy Field *A Aplus Amult Aone Azero Aopp Aeq Ainv Rth Tinvl*

where *A* is a term of type Type, *Aplus* is a term of type A->A->A, *Amult* is a term of type A->A->A, *Aone* is a term of type A, *Azero* is a term of type A, *Aopp* is a term of type A->A, *Aeq* is a term of type A->bool, *Ainv* is a term of type A->A, *Rth* is a term of type (Ring_Theory *A Aplus Amult Aone Azero Ainv Aeq*) , and *Tinvl* is a term of type forall n:*A*, ~(n=*Azero*)->(*Amult* (*Ainv* n) n)=*Aone*. To build a ring theory, refer to Chapter 23 for more details.

This command adds also an entry in the ring theory table if this theory is not already declared. So, it is useless to keep, for a given type, the Add Ring command if you declare a theory with Add Field, except if you plan to use specific features of ring (see Chapter 23). However, the module ring is not loaded by Add Field and you have to make a Require Import Ring if you want to call the ring tactic.

Variants:

1. Add Legacy Field *A Aplus Amult Aone Azero Aopp Aeq Ainv Rth Tinvl*
   with minus:=*Aminus*

   Adds also the term *Aminus* which must be a constant expressed by means of *Aopp*.

2. Add Legacy Field *A Aplus Amult Aone Azero Aopp Aeq Ainv Rth Tinvl*
   with div:=*Adiv*

   Adds also the term *Adiv* which must be a constant expressed by means of *Ainv*.

See also: [42] for more details regarding the implementation of legacy field.

23.10 History of ring

First Samuel Boutin designed the tactic ACDSimpl. This tactic did lot of rewriting. But the proofs terms generated by rewriting were too big for Coq's type-checker. Let us see why:

```
Coq < Goal forall x y z:Z, x + 3 + y + y * z = x + 3 + y + z * y.
1 subgoal

  ============================
   forall x y z : Z, x + 3 + y + y * z = x + 3 + y + z * y

Coq < intros; rewrite (Zmult_comm y z); reflexivity.

Coq < Save toto.

Coq < Print  toto.
toto =
fun x y z : Z =>
eq_ind_r (fun z0 : Z => x + 3 + y + z0 = x + 3 + y + z * y)
  (refl_equal (x + 3 + y + z * y)) (Zmult_comm y z)
     : forall x y z : Z, x + 3 + y + y * z = x + 3 + y + z * y
Argument scopes are [Z_scope Z_scope Z_scope]
```

At each step of rewriting, the whole context is duplicated in the proof term. Then, a tactic that does hundreds of rewriting generates huge proof terms. Since `ACDSimpl` was too slow, Samuel Boutin rewrote it using reflection (see his article in TACS'97 [18]). Later, the stuff was rewritten by Patrick Loiseleur: the new tactic does not any more require `ACDSimpl` to compile and it makes use of $\beta\delta\iota$-reduction not only to replace the rewriting steps, but also to achieve the interleaving of computation and reasoning (see 23.11). He also wrote a few ML code for the `Add Ring` command, that allow to register new rings dynamically.

Proofs terms generated by `ring` are quite small, they are linear in the number of $\oplus$ and $\otimes$ operations in the normalized terms. Type-checking those terms requires some time because it makes a large use of the conversion rule, but memory requirements are much smaller.

23.11 Discussion

Efficiency is not the only motivation to use reflection here. `ring` also deals with constants, it rewrites for example the expression $34+2*x-x+12$ to the expected result $x+46$. For the tactic `ACDSimpl`, the only constants were 0 and 1. So the expression $34+2*(x-1)+12$ is interpreted as $V_0 \oplus V_1 \otimes (V_2 \ominus 1) \oplus V_3$, with the variables mapping $\{V_0 \mapsto 34; V_1 \mapsto 2; V_2 \mapsto x; V_3 \mapsto 12\}$. Then it is rewritten to $34-x+2*x+12$, very far from the expected result. Here rewriting is not sufficient: you have to do some kind of reduction (some kind of *computation*) to achieve the normalization.

The tactic `ring` is not only faster than a classical one: using reflection, we get for free integration of computation and reasoning that would be very complex to implement in the classic fashion.

Is it the ultimate way to write tactics? The answer is: yes and no. The `ring` tactic uses intensively the conversion rule of pCIC, that is replaces proof by computation the most as it is possible. It can be useful in all situations where a classical tactic generates huge proof terms. Symbolic Processing and Tautologies are in that case. But there are also tactics like `auto` or `linear` that do many complex computations, using side-effects and backtracking, and generate a small proof term. Clearly, it would be significantly less efficient to replace them by tactics using reflection.

Another idea suggested by Benjamin Werner: reflection could be used to couple an external tool (a rewriting program or a model checker) with COQ. We define (in COQ) a type of terms, a type of *traces*, and prove a correction theorem that states that *replaying traces* is safe w.r.t some interpretation. Then we let the external tool do every computation (using side-effects, backtracking, exception, or others features that are not available in pure lambda calculus) to produce the trace: now we can check in Coq that the trace has the expected semantic by applying the correction lemma.

Chapter 24

User defined equalities and relations

Matthieu Sozeau

This chapter presents the extension of several equality related tactics to work over user-defined structures (called setoids) that are equipped with ad-hoc equivalence relations meant to behave as equalities. Actually, the tactics have also been generalized to relations weaker then equivalences (e.g. rewriting systems).

This documentation is adapted from the previous setoid documentation by Claudio Sacerdoti Coen (based on previous work by Clément Renard). The new implementation is a drop-in replacement for the old one [1], hence most of the documentation still applies.

The work is a complete rewrite of the previous implementation, based on the type class infrastructure. It also improves on and generalizes the previous implementation in several ways:

- User-extensible algorithm. The algorithm is separated in two parts: generations of the rewriting constraints (done in ML) and solving of these constraints using type class resolution. As type class resolution is extensible using tactics, this allows users to define general ways to solve morphism constraints.

- Sub-relations. An example extension to the base algorithm is the ability to define one relation as a subrelation of another so that morphism declarations on one relation can be used automatically for the other. This is done purely using tactics and type class search.

- Rewriting under binders. It is possible to rewrite under binders in the new implementation, if one provides the proper morphisms. Again, most of the work is handled in the tactics.

- First-class morphisms and signatures. Signatures and morphisms are ordinary Coq terms, hence they can be manipulated inside Coq, put inside structures and lemmas about them can be proved inside the system. Higher-order morphisms are also allowed.

- Performance. The implementation is based on a depth-first search for the first solution to a set of constraints which can be as fast as linear in the size of the term, and the size of the proof term is linear in the size of the original term. Besides, the extensibility allows the user to customize the proof-search if necessary.

[1] Nicolas Tabareau helped with the glueing

24.1 Relations and morphisms

A parametric *relation* R is any term of type `forall (`x_1`:`T_1`) ... (`x_n`:`T_n`), relation` A. The expression A, which depends on $x_1 \ldots x_n$, is called the *carrier* of the relation and R is said to be a relation over A; the list $x_1, \ldots, x_n$ is the (possibly empty) list of parameters of the relation.

Example 1 (Parametric relation) *It is possible to implement finite sets of elements of type A as unordered list of elements of type A. The function `set_eq: forall (A: Type), relation (list A)` satisfied by two lists with the same elements is a parametric relation over `(list A)` with one parameter A. The type of `set_eq` is convertible with `forall (A: Type), list A -> list A -> Prop`.*

An *instance* of a parametric relation R with n parameters is any term `(R` t_1 `...`t_n`)`.

Let R be a relation over A with n parameters. A term is a parametric proof of reflexivity for R if it has type `forall (`x_1`:`T_1`) ... (`x_n`:`T_n`), reflexive (R` x_1 `...`x_n`)`. Similar definitions are given for parametric proofs of symmetry and transitivity.

Example 2 (Parametric relation (cont.)) *The `set_eq` relation of the previous example can be proved to be reflexive, symmetric and transitive.*

A parametric unary function f of type `forall (`x_1`:`T_1`) ... (`x_n`:`T_n`),` A_1 `->` A_2 covariantly respects two parametric relation instances R_1 and R_2 if, whenever x, y satisfy $R_1\ x\ y$, their images $(f\ x)$ and $(f\ y)$ satisfy $R_2\ (f\ x)\ (f\ y)$. An f that respects its input and output relations will be called a unary covariant *morphism*. We can also say that f is a monotone function with respect to R_1 and R_2. The sequence $x_1, \ldots x_n$ represents the parameters of the morphism.

Let R_1 and R_2 be two parametric relations. The *signature* of a parametric morphism of type `forall (`x_1`:`T_1`) ... (`x_n`:`T_n`),` A_1 `->` A_2 that covariantly respects two instances I_{R_1} and I_{R_2} of R_1 and R_2 is written I_{R_1}`++>`I_{R_2}. Notice that the special arrow ++>, which reminds the reader of covariance, is placed between the two relation instances, not between the two carriers. The signature relation instances and morphism will be typed in a context introducing variables for the parameters.

The previous definitions are extended straightforwardly to n-ary morphisms, that are required to be simultaneously monotone on every argument.

Morphisms can also be contravariant in one or more of their arguments. A morphism is contravariant on an argument associated to the relation instance R if it is covariant on the same argument when the inverse relation R^{-1} (`inverse R` in Coq) is considered. The special arrow `-->` is used in signatures for contravariant morphisms.

Functions having arguments related by symmetric relations instances are both covariant and contravariant in those arguments. The special arrow `==>` is used in signatures for morphisms that are both covariant and contravariant.

An instance of a parametric morphism f with n parameters is any term `f` t_1 `...`t_n.

Example 3 (Morphisms) *Continuing the previous example, let `union: forall (A: Type), list A -> list A -> list A` perform the union of two sets by appending one list to the other. `union` is a binary morphism parametric over A that respects the relation instance `(set_eq A)`. The latter condition is proved by showing `forall (A: Type) (S1 S1' S2 S2': list A), set_eq A S1 S1' -> set_eq A S2 S2' -> set_eq A (union A S1 S2) (union A S1' S2')`.*

The signature of the function `union A` is `set_eq A ==> set_eq A ==> set_eq A` for all A.

Example 4 (Contravariant morphism) *The division function* `Rdiv: R -> R -> R` *is a morphism of signature* `le ++> le --> le` *where* `le` *is the usual order relation over real numbers. Notice that division is covariant in its first argument and contravariant in its second argument.*

Leibniz equality is a relation and every function is a morphism that respects Leibniz equality. Unfortunately, Leibniz equality is not always the intended equality for a given structure.

In the next section we will describe the commands to register terms as parametric relations and morphisms. Several tactics that deal with equality in Coq can also work with the registered relations. The exact list of tactic will be given in Sect. 24.7. For instance, the tactic `reflexivity` can be used to close a goal $R\ n\ n$ whenever R is an instance of a registered reflexive relation. However, the tactics that replace in a context $C[]$ one term with another one related by R must verify that $C[]$ is a morphism that respects the intended relation. Currently the verification consists in checking whether $C[]$ is a syntactic composition of morphism instances that respects some obvious compatibility constraints.

Example 5 (Rewriting) *Continuing the previous examples, suppose that the user must prove* `set_eq int (union int (union int S1 S2) S2) (f S1 S2)` *under the hypothesis* `H: set_eq int S2 (nil int)`*. It is possible to use the* `rewrite` *tactic to replace the first two occurrences of* `S2` *with* `nil int` *in the goal since the context* `set_eq int (union int (union int S1 nil) nil) (f S1 S2)`*, being a composition of morphisms instances, is a morphism. However the tactic will fail replacing the third occurrence of* `S2` *unless* `f` *has also been declared as a morphism.*

24.2 Adding new relations and morphisms

A parametric relation *Aeq*: `forall` $(y_1 : \beta_!\ \ldots y_m : \beta_m)$`,` `relation (A` $t_1\ \ldots t_n$`)` over (*A :* α_i `->` $\ldots\ \alpha_n$ `->` `Type`) can be declared with the following command:

`Add Parametric Relation` $(x_1 : T_1) \ldots (x_n : T_k)$: *(A* $t_1 \ldots t_n$*) (Aeq* $t'_1 \ldots t'_m$*)*
[`reflexivity proved by` *refl]*
[`symmetry proved by` *sym]*
[`transitivity proved by` *trans]*
`as` *id.*

after having required the `Setoid` module with the `Require Setoid` command.

The identifier *id* gives a unique name to the morphism and it is used by the command to generate fresh names for automatically provided lemmas used internally.

Notice that the carrier and relation parameters may refer to the context of variables introduced at the beginning of the declaration, but the instances need not be made only of variables. Also notice that *A* is *not* required to be a term having the same parameters as *Aeq*, although that is often the case in practice (this departs from the previous implementation).

In case the carrier and relations are not parametric, one can use the command `Add Relation` instead, whose syntax is the same except there is no local context.

The proofs of reflexivity, symmetry and transitivity can be omitted if the relation is not an equivalence relation. The proofs must be instances of the corresponding relation definitions: e.g. the proof of reflexivity must have a type convertible to `reflexive (A` $t_1\ \ldots t_n$`)` `(Aeq` $t'_1\ \ldots t'_n$`)`. Each proof may refer to the introduced variables as well.

Example 6 (Parametric relation) *For leibniz equality, we may declare:* `Add Parametric Relation (A : Type) : A (@eq A)`
`[reflexivity proved by @refl_equal A]`
...

Some tactics (`reflexivity`, `symmetry`, `transitivity`) work only on relations that respect the expected properties. The remaining tactics (`replace`, `rewrite` and derived tactics such as `autorewrite`) do not require any properties over the relation. However, they are able to replace terms with related ones only in contexts that are syntactic compositions of parametric morphism instances declared with the following command.

`Add Parametric Morphism` $(x_1 : \mathrm{T}_!) \ldots (x_k : \mathrm{T}_k)$
$(f\, t_1 \ldots t_n)$
`with signature` *sig*
`as id.`
`Proof`
...
`Qed`

The command declares f as a parametric morphism of signature *sig*. The identifier *id* gives a unique name to the morphism and it is used as the base name of the type class instance definition and as the name of the lemma that proves the well-definedness of the morphism. The parameters of the morphism as well as the signature may refer to the context of variables. The command asks the user to prove interactively that f respects the relations identified from the signature.

Example 7 *We start the example by assuming a small theory over homogeneous sets and we declare set equality as a parametric equivalence relation and union of two sets as a parametric morphism.*

```
Coq < Require Export Setoid.

Coq < Require Export Relation_Definitions.

Coq < Set Implicit Arguments.

Coq < Parameter set: Type -> Type.

Coq < Parameter empty: forall A, set A.

Coq < Parameter eq_set: forall A, set A -> set A -> Prop.

Coq < Parameter union: forall A, set A -> set A -> set A.

Coq < Axiom eq_set_refl: forall A, reflexive _ (eq_set (A:=A)).

Coq < Axiom eq_set_sym: forall A, symmetric _ (eq_set (A:=A)).

Coq < Axiom eq_set_trans: forall A, transitive _ (eq_set (A:=A)).

Coq < Axiom empty_neutral: forall A (S: set A), eq_set (union S (empty A)) S.

Coq < Axiom union_compat:
Coq <  forall (A : Type),
Coq <   forall x x' : set A, eq_set x x' ->
Coq <   forall y y' : set A, eq_set y y' ->
Coq <    eq_set (union x y) (union x' y').

Coq < Add Parametric Relation A : (set A) (@eq_set A)
```

```
Coq <  reflexivity proved by (eq_set_refl (A:=A))
Coq <  symmetry proved by (eq_set_sym (A:=A))
Coq <  transitivity proved by (eq_set_trans (A:=A))
Coq <  as eq_set_rel.

Coq < Add Parametric Morphism A : (@union A) with
Coq < signature (@eq_set A) ==> (@eq_set A) ==> (@eq_set A) as union_mor.

Coq < Proof. exact (@union_compat A). Qed.
```

Is is possible to reduce the burden of specifying parameters using (maximally inserted) implicit arguments. If `A` is always set as maximally implicit in the previous example, one can write:

```
Coq < Add Parametric Relation A : (set A) eq_set
Coq <  reflexivity proved by eq_set_refl
Coq <  symmetry proved by eq_set_sym
Coq <  transitivity proved by eq_set_trans
Coq <  as eq_set_rel.

Coq < Add Parametric Morphism A : (@union A) with
Coq <   signature eq_set ==> eq_set ==> eq_set as union_mor.

Coq < Proof. exact (@union_compat A). Qed.
```

We proceed now by proving a simple lemma performing a rewrite step and then applying reflexivity, as we would do working with Leibniz equality. Both tactic applications are accepted since the required properties over `eq_set` and `union` can be established from the two declarations above.

```
Coq < Goal forall (S: set nat),
Coq <  eq_set (union (union S empty) S) (union S S).

Coq < Proof. intros. rewrite empty_neutral. reflexivity. Qed.
```

The tables of relations and morphisms are managed by the type class instance mechanism. The behavior on section close is to generalize the instances by the variables of the section (and possibly hypotheses used in the proofs of instance declarations) but not to export them in the rest of the development for proof search. One can use the `Existing Instance` command to do so outside the section, using the name of the declared morphism suffixed by `_Morphism`, or use the `Global` modifier for the corresponding class instance declaration (see §24.6) at definition time. When loading a compiled file or importing a module, all the declarations of this module will be loaded.

24.3 Rewriting and non reflexive relations

To replace only one argument of an n-ary morphism it is necessary to prove that all the other arguments are related to themselves by the respective relation instances.

Example 8 *To replace* `(union S empty)` *with* `S` *in* `(union (union S empty) S) (union S S)` *the rewrite tactic must exploit the monotony of* `union` *(axiom* `union_compat` *in the previous example). Applying* `union_compat` *by hand we are left with the goal* `eq_set (union S S) (union S S)`.

When the relations associated to some arguments are not reflexive, the tactic cannot automatically prove the reflexivity goals, that are left to the user.

Setoids whose relation are partial equivalence relations (PER) are useful to deal with partial functions. Let `R` be a PER. We say that an element `x` is defined if `R x x`. A partial function whose domain comprises all the defined elements only is declared as a morphism that respects `R`. Every time a rewriting step is performed the user must prove that the argument of the morphism is defined.

Example 9 *Let `eq0` be `fun x y => x = y ∧ x≠ 0` (the smaller PER over non zero elements). Division can be declared as a morphism of signature `eq ==> eq0 ==> eq`. Replace `x` with `y` in `div x n = div y n` opens the additional goal `eq0 n n` that is equivalent to `n=n ∧ n≠0`.*

24.4 Rewriting and non symmetric relations

When the user works up to relations that are not symmetric, it is no longer the case that any covariant morphism argument is also contravariant. As a result it is no longer possible to replace a term with a related one in every context, since the obtained goal implies the previous one if and only if the replacement has been performed in a contravariant position. In a similar way, replacement in an hypothesis can be performed only if the replaced term occurs in a covariant position.

Example 10 (Covariance and contravariance) *Suppose that division over real numbers has been defined as a morphism of signature `Zdiv: Zlt ++> Zlt --> Zlt` (i.e. `Zdiv` is increasing in its first argument, but decreasing on the second one). Let < denotes `Zlt`. Under the hypothesis `H: x < y` we have `k < x / y -> k < x / x`, but not `k < y / x -> k < x / x`. Dually, under the same hypothesis `k < x / y -> k < y / y` holds, but `k < y / x -> k < y / y` does not. Thus, if the current goal is `k < x / x`, it is possible to replace only the second occurrence of `x` (in contravariant position) with `y` since the obtained goal must imply the current one. On the contrary, if `k < x / x` is an hypothesis, it is possible to replace only the first occurrence of `x` (in covariant position) with `y` since the current hypothesis must imply the obtained one.*

Contrary to the previous implementation, no specific error message will be raised when trying to replace a term that occurs in the wrong position. It will only fail because the rewriting constraints are not satisfiable. However it is possible to use the `at` modifier to specify which occurences should be rewritten.

As expected, composing morphisms together propagates the variance annotations by switching the variance every time a contravariant position is traversed.

Example 11 *Let us continue the previous example and let us consider the goal `x / (x / x) < k`. The first and third occurrences of `x` are in a contravariant position, while the second one is in covariant position. More in detail, the second occurrence of `x` occurs covariantly in `(x / x)` (since division is covariant in its first argument), and thus contravariantly in `x / (x / x)` (since division is contravariant in its second argument), and finally covariantly in `x / (x / x) < k` (since <, as every transitive relation, is contravariant in its first argument with respect to the relation itself).*

24.5 Rewriting in ambiguous setoid contexts

One function can respect several different relations and thus it can be declared as a morphism having multiple signatures.

Example 12 *Union over homogeneous lists can be given all the following signatures:* `eq ==> eq ==> eq` *(*`eq` *being the equality over ordered lists)* `set_eq ==> set_eq ==> set_eq` *(*`set_eq` *being the equality over unordered lists up to duplicates),* `multiset_eq ==> multiset_eq ==> multiset_eq` *(*`multiset_eq` *being the equality over unordered lists).*

To declare multiple signatures for a morphism, repeat the `Add Morphism` command.

When morphisms have multiple signatures it can be the case that a rewrite request is ambiguous, since it is unclear what relations should be used to perform the rewriting. Contrary to the previous implementation, the tactic will always choose the first possible solution to the set of constraints generated by a rewrite and will not try to find *all* possible solutions to warn the user about.

24.6 First class setoids and morphisms

The implementation is based on a first-class representation of properties of relations and morphisms as type classes. That is, the various combinations of properties on relations and morphisms are represented as records and instances of theses classes are put in a hint database. For example, the declaration:

> `Add Parametric Relation` $(x_1 : T_1) \ldots (x_n : T_k) : (A\ t_1 \ldots t_n)\ (Aeq\ t'_1 \ldots t'_m)$
> *[*`reflexivity proved by` *refl]*
> *[*`symmetry proved by` *sym]*
> *[*`transitivity proved by` *trans]*
> `as` *id.*

is equivalent to an instance declaration:

> `Instance` $(x_1 : T_1) \ldots (x_n : T_k)$ => *id* : `@Equivalence` $(A\ t_1 \ldots t_n)\ (Aeq\ t'_1 \ldots t'_m)$:=
> *[*`Equivalence_Reflexive :=` *refl]*
> *[*`Equivalence_Symmetric :=` *sym]*
> *[*`Equivalence_Transitive :=` *trans].*

The declaration itself amounts to the definition of an object of the record type `Coq.Classes.RelationClasses.Equivalence` and a hint added to the `typeclass_instances` hint database. Morphism declarations are also instances of a type class defined in `Classes.Morphisms`. See the documentation on type classes 18 and the theories files in `Classes` for further explanations.

One can inform the rewrite tactic about morphisms and relations just by using the typeclass metchanism to declare them using `Instance` and `Context` vernacular commands. Any object of type `Morphism` in the local context will also be automatically used by the rewriting tactic to solve constraints.

Other representations of first class setoids and morphisms can also be handled by encoding them as records. In the following example, the projections of the setoid relation and of the morphism function can be registered as parametric relations and morphisms.

Example 13 (First class setoids) `Coq < Require Import Relation_Definitions Setoid.`

```
Coq < Record Setoid: Type :=
Coq < { car:Type;
Coq <   eq:car->car->Prop;
Coq <   refl: reflexive _ eq;
```

```
Coq <   sym: symmetric _ eq;
Coq <   trans: transitive _ eq
Coq < }.

Coq < Add Parametric Relation (s : Setoid) : (@car s) (@eq s)
Coq <  reflexivity proved by (refl s)
Coq <  symmetry proved by (sym s)
Coq <  transitivity proved by (trans s) as eq_rel.

Coq < Record Morphism (S1 S2:Setoid): Type :=
Coq < { f:car S1 ->car S2;
Coq <   compat: forall (x1 x2: car S1), eq S1 x1 x2 -> eq S2 (f x1) (f x2) }.

Coq < Add Parametric Morphism (S1 S2 : Setoid) (M : Morphism S1 S2) :
Coq <  (@f S1 S2 M) with signature (@eq S1 ==> @eq S2) as apply_mor.

Coq < Proof. apply (compat S1 S2 M). Qed.

Coq < Lemma test: forall (S1 S2:Setoid) (m: Morphism S1 S2)
Coq <  (x y: car S1), eq S1 x y -> eq S2 (f _ _ m x) (f _ _ m y).

Coq < Proof. intros. rewrite H. reflexivity. Qed.
```

24.7 Tactics enabled on user provided relations

The following tactics, all prefixed by `setoid_`, deal with arbitrary registered relations and morphisms. Moreover, all the corresponding unprefixed tactics (i.e. `reflexivity`, `symmetry`, `transitivity`, `replace`, `rewrite`) have been extended to fall back to their prefixed counterparts when the relation involved is not Leibniz equality. Notice, however, that using the prefixed tactics it is possible to pass additional arguments such as `using relation`.

`setoid_reflexivity`
`setoid_symmetry` *[*`in` *ident]*
`setoid_transitivity`
`setoid_rewrite` *[orientation] term [*`at` *occs] [*`in` *ident]*
`setoid_replace` *term* `with` *term [*`in` *ident] [*`using relation` *term] [*`by` *tactic]*

The `using relation` arguments cannot be passed to the unprefixed form. The latter argument tells the tactic what parametric relation should be used to replace the first tactic argument with the second one. If omitted, it defaults to the `DefaultRelation` instance on the type of the objects. By default, it means the most recent `Equivalence` instance in the environment, but it can be customized by declaring new `DefaultRelation` instances. As leiniz equality is a declared equivalence, it will fall back to it if no other relation is declared on a type.

Every derived tactic that is based on the unprefixed forms of the tactics considered above will also work up to user defined relations. For instance, it is possible to register hints for `autorewrite` that are not proof of Leibniz equalities. In particular it is possible to exploit `autorewrite` to simulate normalization in a term rewriting system up to user defined equalities.

24.8 Printing relations and morphisms

The `Print Instances` command can be used to show the list of currently registered `Reflexive` (using `Print Instances Reflexive`), `Symmetric` or `Transitive` relations, `Equivalence`s, `PreOrder`s, `PER`s, and `Morphism`s. When the rewriting tactics refuse to replace

a term in a context because the latter is not a composition of morphisms, the `Print Instances` commands can be useful to understand what additional morphisms should be registered.

24.9 Deprecated syntax and backward incompatibilities

Due to backward compatibility reasons, the following syntax for the declaration of setoids and morphisms is also accepted.

`Add Setoid` *A Aeq ST* `as` *ident*

where *Aeq* is a congruence relation without parameters, *A* is its carrier and *ST* is an object of type `(Setoid_Theory A Aeq)` (i.e. a record packing together the reflexivity, symmetry and transitivity lemmas). Notice that the syntax is not completely backward compatible since the identifier was not required.

`Add Morphism` *f*:*ident*.
Proof.
...
Qed.

The latter command also is restricted to the declaration of morphisms without parameters. It is not fully backward compatible since the property the user is asked to prove is slightly different: for n-ary morphisms the hypotheses of the property are permuted; moreover, when the morphism returns a proposition, the property is now stated using a bi-implication in place of a simple implication. In practice, porting an old development to the new semantics is usually quite simple.

Notice that several limitations of the old implementation have been lifted. In particular, it is now possible to declare several relations with the same carrier and several signatures for the same morphism. Moreover, it is now also possible to declare several morphisms having the same signature. Finally, the replace and rewrite tactics can be used to replace terms in contexts that were refused by the old implementation. As discussed in the next section, the semantics of the new `setoid_rewrite` command differs slightly from the old one and `rewrite`.

24.10 Rewriting under binders

Warning: Due to compatibility issues, this feature is enabled only when calling the `setoid_rewrite` tactics directly and not `rewrite`.

To be able to rewrite under binding constructs, one must declare morphisms with respect to pointwise (setoid) equivalence of functions. Example of such morphisms are the standard `all` and `ex` combinators for universal and existential quantification respectively. They are declared as morphisms in the `Classes.Morphisms_Prop` module. For example, to declare that universal quantification is a morphism for logical equivalence:

```
Coq < Instance all_iff_morphism (A : Type) :
Coq <   Morphism (pointwise_relation A iff ==> iff) (@all A).
Toplevel input, characters 51-83:
>   Morphism (pointwise_relation A iff ==> iff) (@all A).
>              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: Unknown interpretation for notation "_ ==> _".
```

```
Coq < Proof. simpl_relation.
Error: Unknown command of the non proof-editing mode.
Error: Unknown command of the non proof-editing mode.
```

One then has to show that if two predicates are equivalent at every point, their universal quantifications are equivalent. Once we have declared such a morphism, it will be used by the setoid rewriting tactic each time we try to rewrite under an `all` application (products in Prop are implicitely translated to such applications).

Indeed, when rewriting under a lambda, binding variable x, say from $P\ x$ to $Q\ x$ using the relation `iff`, the tactic will generate a proof of `pointwise_relation A iff (fun x => P x) (fun x => Q x)` from the proof of `iff (P x) (Q x)` and a constraint of the form `Morphism (pointwise_relation A iff ==> ?) m` will be generated for the surrounding morphism `m`.

Hence, one can add higher-order combinators as morphisms by providing signatures using pointwise extension for the relations on the functional arguments (or whatever subrelation of the pointwise extension). For example, one could declare the `map` combinator on lists as a morphism:

```
Coq < Instance map_morphism `{Equivalence A eqA, Equivalence B eqB} :
Coq <   Morphism ((eqA ==> eqB) ==> list_equiv eqA ==> list_equiv eqB)
Coq <       (@map A B).
```

where `list_equiv` implements an equivalence on lists parameterized by an equivalence on the elements.

Note that when one does rewriting with a lemma under a binder using `setoid_rewrite`, the application of the lemma may capture the bound variable, as the semantics are different from rewrite where the lemma is first matched on the whole term. With the new `setoid_rewrite`, matching is done on each subterm separately and in its local environment, and all matches are rewritten *simultaneously* by default. The semantics of the previous `setoid_rewrite` implementation can almost be recovered using the `at 1` modifier.

24.11 Sub-relations

Sub-relations can be used to specify that one relation is included in another, so that morphisms signatures for one can be used for the other. If a signature mentions a relation R on the left of an arrow `==>`, then the signature also applies for any relation S that is smaller than R, and the inverse applies on the right of an arrow. One can then declare only a few morphisms instances that generate the complete set of signatures for a particular constant. By default, the only declared subrelation is `iff`, which is a subrelation of `impl` and `inverse impl` (the dual of implication). That's why we can declare only two morphisms for conjunction: `Morphism (impl ==> impl ==> impl) and` and `Morphism (iff ==> iff ==> iff) and`. This is sufficient to satisfy any rewriting constraints arising from a rewrite using `iff`, `impl` or `inverse impl` through `and`.

Sub-relations are implemented in `Classes.Morphisms` and are a prime example of a mostly user-space extension of the algorithm.

24.12 Constant unfolding

The resolution tactic is based on type classes and hence regards user-defined constants as transparent by default. This may slow down the resolution due to a lot of unifications (all the declared `Morphism` instances are tried at each node of the search tree). To speed it up, declare your constant as rigid for proof search using the command `Typeclasses Opaque` (see §18.5.4).

Chapter 25

Calling external provers

25.1 The gappa tactic

Sylvie Boldo, Guillaume Melquiond, Jean-Christophe Filliâtre

The `gappa` tactic invokes the Gappa tool[1] to solve properties about floating-point or fixed-point arithmetic. It can also solve simple inequalities over real numbers.

The Gappa tool must be installed and its executable (called `gappa`) must be in the user program path. The Coq support library for Gappa must also be installed (it is available from Gappa's web site). This library provides a `Gappa_tactic` module, which must be loaded for the tactic to work properly.

The `gappa` tactic only handles goals and hypotheses that are double inequalities $f_1 \leq e \leq f_2$ where f_1 and f_2 are dyadic constants and e a real-valued expression. Here is an example of a goal solved by `gappa`:

```
Lemma test_gappa :
  forall x y:R,
  3/4 <= x <= 3 ->
  0 <= sqrt x <= 1775 * (powerRZ 2 (-10)).
Proof.
  gappa.
Qed.
```

Gappa supports floating-point rounding operations (as functions over real numbers). Here is an example involving double-precision floating-point numbers with rounding toward zero:

```
Definition rnd := gappa_rounding (rounding_float roundZR 53 1074).

Lemma test_gappa2 :
  forall a_ b_ a b : R,
  a = rnd a_ ->
  b = rnd b_ ->
  52 / 16 <= a <= 53 / 16 ->
```

[1] http://lipforge.ens-lyon.fr/www/gappa/

```
  22 / 16 <= b <= 30 / 16 ->
  0 <= rnd (a - b) - (a - b) <= 0.
Proof.
  unfold rnd; gappa.
Qed.
```

The function `gappa_rounding` declares a rounding mode recognized by the `gappa` tactic. Rounding modes are built using constants such as `rounding_float` and `roundZR` provided by the Gappa support library.

Bibliography

[1] David Aspinall. Proof general. http://proofgeneral.inf.ed.ac.uk/.

[2] Ph. Audebaud. Partial Objects in the Calculus of Constructions. In *Proceedings of the sixth Conf. on Logic in Computer Science*. IEEE, 1991.

[3] Ph. Audebaud. CC+ : an extension of the Calculus of Constructions with fixpoints. In B. Nordström and K. Petersson and G. Plotkin, editor, *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 21–34, 1992. Also Research Report LIP-ENS-Lyon.

[4] Ph. Audebaud. *Extension du Calcul des Constructions par Points fixes*. PhD thesis, Université Bordeaux I, 1992.

[5] L. Augustsson. Compiling Pattern Matching. In *Conference Functional Programming and Computer Architecture*, 1985.

[6] H. Barendregt. Lambda Calculi with Types. Technical Report 91-19, Catholic University Nijmegen, 1991. In Handbook of Logic in Computer Science, Vol II.

[7] H. Barendregt and T. Nipkow, editors. *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[8] H.P. Barendregt. *The Lambda Calculus its Syntax and Semantics*. North-Holland, 1981.

[9] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, 1999.

[10] J.L. Bates and R.L. Constable. Proofs as Programs. *ACM transactions on Programming Languages and Systems*, 7, 1985.

[11] M.J. Beeson. *Foundations of Constructive Mathematics, Metamathematical Studies*. Springer-Verlag, 1985.

[12] 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.

[13] Stefano Berardi and Mario Coppo, editors. *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*. Springer, 1996.

[14] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS series. Springer Verlag, 2004.

[15] E. Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, 1967.

[16] S. Boutin. Certification d'un compilateur ML en Coq. Master's thesis, Université Paris 7, September 1992.

[17] S. Boutin. *Réflexions sur les quotients*. thèse d'université, Paris 7, April 1997.

[18] S. Boutin. Using reflection to build efficient and certified decision procedure s. In Martin Abadi and Takahashi Ito, editors, *TACS'97*, volume 1281 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[19] R.S. Boyer and J.S. Moore. *A computational logic*. ACM Monograph. Academic Press, 1979.

[20] Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors. *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers*, volume 2277 of *Lecture Notes in Computer Science*. Springer, 2002.

[21] Laurent Chicli, Loïc Pottier, and Carlos Simpson. Mathematical quotients and quotient types in coq. In Geuvers and Wiedijk [64].

[22] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[23] Th. Coquand. *Une Théorie des Constructions*. PhD thesis, Université Paris 7, January 1985.

[24] Th. Coquand. An Analysis of Girard's Paradox. In *Symposium on Logic in Computer Science*, Cambridge, MA, 1986. IEEE Computer Society Press.

[25] Th. Coquand. Metamathematical Investigations of a Calculus of Constructions. In P. Oddifredi, editor, *Logic and Computer Science*. Academic Press, 1990. INRIA Research Report 1088, also in [62].

[26] Th. Coquand. A New Paradox in Type Theory. In *Proceedings 9th Int. Congress of Logic, Methodology and Philosophy of Science*, August 1991.

[27] Th. Coquand. Pattern Matching with Dependent Types. In Nordström et al. [106].

[28] Th. Coquand. Infinite Objects in Type Theory. In Barendregt and Nipkow [7].

[29] Th. Coquand and G. Huet. Constructions : A Higher Order Proof System for Mechanizing Mathematics. In *EUROCAL'85*, volume 203 of *Lecture Notes in Computer Science*, Linz, 1985. Springer-Verlag.

[30] 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.

[31] Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.

[32] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[33] P. Corbineau. A declarative language for the coq proof assistant. In M. Miculan, I. Scagnetto, and F. Honsell, editors, *TYPES '07, Cividale del Friuli, Revised Selected Papers*, volume 4941 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2007.

[34] C. Cornes. *Conception d'un langage de haut niveau de représentation de preuves*. Thèse de doctorat, Université Paris 7, November 1997.

[35] Cristina Cornes and Delphine Terrasse. Automating inversion of inductive predicates in coq. In Berardi and Coppo [13], pages 85–104.

[36] J. Courant. Explicitation de preuves par récurrence implicite. Master's thesis, DEA d'Informatique, ENS Lyon, September 1994.

[37] 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.

[38] 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.

[39] D. de Rauglaudre. Camlp4 version 1.07.2. In Camlp4 distribution, 1998.

[40] D. Delahaye. Information retrieval in a coq proof library using type isomorphisms. In *Proceedings of TYPES '99, Lökeberg*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[41] D. Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer-Verlag, November 2000.

[42] D. Delahaye and M. Mayero. `Field`: une procédure de décision pour les nombres réels en COQ. In *Journées Francophones des Langages Applicatifs, Pontarlier*. INRIA, Janvier 2001.

[43] R. di Cosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design*. Progress in Theoretical Computer Science. Birkhauser, 1995. ISBN-0-8176-3763-X.

[44] G. Dowek. Naming and scoping in a mathematical vernacular. Research Report 1283, INRIA, 1990.

[45] G. Dowek. *Démonstration automatique dans le Calcul des Constructions*. PhD thesis, Université Paris 7, December 1991.

[46] G. Dowek. L'indécidabilité du filtrage du troisième ordre dans les calculs avec types dépendants ou constructeurs de types. *Compte-Rendus 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.

[47] 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. Springer-Verlag, 1991. Also INRIA Research Report.

[48] G. Dowek. A Complete Proof Synthesis Method for the Cube of Type Systems. *Journal Logic Computation*, 3(3):287–315, June 1993.

[49] G. Dowek. The undecidability of pattern matching in calculi where primitive recursive functions are representable. *Theoretical Computer Science*, 107(2):349–356, 1993.

[50] G. Dowek. Third order matching is decidable. *Annals of Pure and Applied Logic*, 69:135–155, 1994.

[51] G. Dowek. Lambda-calculus, combinators and the comprehension schema. In *Proceedings of the second international conference on typed lambda calculus and applications*, 1995.

[52] 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. Technical Report 154, INRIA, May 1993.

[53] 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.

[54] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3), September 1992.

[55] J.-C. Filliâtre. Une procédure de décision pour le calcul des prédicats direct. Étude et implémentation dans le système Coq. Master's thesis, DEA d'Informatique, ENS Lyon, September 1994.

[56] J.-C. Filliâtre. A decision procedure for direct predicate calculus. Research report 96–25, LIP-ENS-Lyon, 1995.

[57] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université Paris-Sud, July 1999.

[58] J.-C. Filliâtre. Formal Proof of a Program: Find. Submitted to *Science of Computer Programming*, January 2000.

[59] J.-C. Filliâtre and N. Magaud. Certification of sorting algorithms in the system Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.

[60] J.-C. Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, July 2003. [English translation of [57]].

[61] E. Fleury. Implantation des algorithmes de Floyd et de Dijkstra dans le Calcul des Constructions. Rapport de Stage, July 1990.

[62] Projet Formel. The Calculus of Constructions. Documentation and user's guide, Version 4.10. Technical Report 110, INRIA, 1989.

[63] Jean-Baptiste-Joseph Fourier. *Fourier's method to solve linear inequations/equations systems.* Gauthier-Villars, 1890.

[64] H. Geuvers and F. Wiedijk, editors. *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[65] E. Giménez. Codifying guarded definitions with recursive schemes. In *Types'94 : Types for Proofs and Programs*, volume 996 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. Extended version in LIP research report 95-07, ENS Lyon.

[66] E. Giménez. An application of co-inductive types in coq: verification of the alternating bit protocol. In *Workshop on Types for Proofs and Programs*, number 1158 in Lecture Notes in Computer Science, pages 135–152. Springer-Verlag, 1995.

[67] E. Giménez. A tutorial on recursive types in coq. Technical report, INRIA, March 1998.

[68] E. Giménez and P. Castéran. A tutorial on [co-]inductive types in coq. available at `http://coq.inria.fr/doc`, January 2005.

[69] 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 théorie des types. In *Proceedings of the 2nd Scandinavian Logic Symposium*. North-Holland, 1970.

[70] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972.

[71] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989.

[72] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Centre,, 1995.

[73] D. Hirschkoff. Écriture d'une tactique arithmétique pour le système COQ. Master's thesis, DEA IARFA, Ecole des Ponts et Chaussées, Paris, September 1994.

[74] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Proceedings of the meeting Twenty-five years of constructive type theory*. Oxford University Press, 1998.

[75] 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.

[76] G. Huet. Programming of future generation computers. In *Proceedings of TAPSOFT87*, volume 249 of *Lecture Notes in Computer Science*, pages 276–286. Springer-Verlag, 1987.

[77] 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 [76].

[78] G. Huet, editor. *Logical Foundations of Functional Programming*. The UT Year of Programming Series. Addison-Wesley, 1989.

[79] 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 [62].

[80] G. Huet. The gallina specification language : A case study. In *Proceedings of 12th FST/TCS Conference, New Delhi*, volume 652 of *Lecture Notes in Computer Science*, pages 229–240. Springer-Verlag, 1992.

[81] G. Huet. Residual theory in λ-calculus: a formal development. *J. Functional Programming*, 4,3:371–394, 1994.

[82] G. Huet and J.-J. Lévy. Call by need computations in non-ambigous linear term rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*. The MIT press, 1991. Also research report 359, INRIA, 1979.

[83] G. Huet and G. Plotkin, editors. *Logical Frameworks*. Cambridge University Press, 1991.

[84] G. Huet and G. Plotkin, editors. *Logical Environments*. Cambridge University Press, 1992.

[85] J. Ketonen and R. Weyhrauch. A decidable fragment of Predicate Calculus. *Theoretical Computer Science*, 32:297–307, 1984.

[86] S.C. Kleene. *Introduction to Metamathematics*. Bibliotheca Mathematica. North-Holland, 1952.

[87] J.-L. Krivine. *Lambda-calcul types et modèles*. Etudes et recherche en informatique. Masson, 1990.

[88] A. Laville. Comparison of priority rules in pattern matching and term rewriting. *Journal of Symbolic Computation*, 11:321–347, 1991.

[89] 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.

[90] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[91] P. Letouzey. A new extraction for coq. In Geuvers and Wiedijk [64].

[92] L.Puel and A. Suárez. Compiling Pattern Matching by Term Decomposition. In *Conference Lisp and Functional Programming*, ACM. Springer-Verlag, 1990.

[93] Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.

[94] P. Manoury. A User's Friendly Syntax to Define Recursive Functions as Typed $\lambda-$Terms. In *Types for Proofs and Programs, TYPES'94*, volume 996 of *LNCS*, June 1994.

[95] P. Manoury and M. Simonot. Automatizing termination proofs of recursively defined functions. *TCS*, 135(2):319–343, 1994.

[96] L. Maranget. Two Techniques for Compiling Lazy Pattern Matching. Technical Report 2385, INRIA, 1994.

[97] Conor McBride. Elimination with a motive. In Callaghan et al. [20], pages 197–216.

[98] A. Miquel. A model for impredicative type systems with universes, intersection types and subtyping. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science (LICS'00)*. IEEE Computer Society Press, 2000.

[99] A. Miquel. The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In *Proceedings of the fifth International Conference on Typed Lambda Calculi and Applications (TLCA01), Krakow, Poland*, number 2044 in LNCS. Springer-Verlag, 2001.

[100] A. Miquel. *Le Calcul des Constructions implicite: syntaxe et sémantique*. PhD thesis, Université Paris 7, dec 2001.

[101] A. Miquel and B. Werner. The not so simple proof-irrelevant model of cc. In Geuvers and Wiedijk [64], pages 240–258.

[102] C. Muñoz. *Un calcul de substitutions pour la représentation de preuves partielles en théorie de types*. Thèse de doctorat, Université Paris 7, 1997. Version en anglais disponible comme rapport de recherche INRIA RR-3309.

[103] C. Muñoz. Démonstration automatique dans la logique propositionnelle intuitionniste. Master's thesis, DEA d'Informatique Fondamentale, Université Paris 7, September 1994.

[104] B. Nordström. Terminating general recursion. *BIT*, 28, 1988.

[105] 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.

[106] B. Nordström, 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.

[107] P. Odifreddi, editor. *Logic and Computer Science*. Academic Press, 1990.

[108] P. Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, 1984.

[109] 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 [7].

[110] C. Parent. *Synthèse de preuves de programmes dans le Calcul des Constructions Inductives*. PhD thesis, Ecole Normale Supérieure de Lyon, 1995.

[111] C. Parent. Synthesizing proofs from programs in the Calculus of Inductive Constructions. In *Mathematics of Program Construction'95*, volume 947 of *LNCS*. Springer-Verlag, 1995.

[112] M. Parigot. Recursive Programming with Proofs. *Theoretical Computer Science*, 94(2):335–356, 1992.

[113] 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.

[114] 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.

[115] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Université Paris 7, January 1989.

[116] 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. Springer-Verlag, 1993. Also LIP research report 92-49, ENS Lyon.

[117] C. Paulin-Mohring. *Le système Coq. Thèse d'habilitation*. ENS Lyon, January 1997.

[118] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[119] K.V. Prasad. Programming with broadcasts. In *Proceedings of CONCUR'93*, volume 715 of *LNCS*. Springer-Verlag, 1993.

[120] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communication of the ACM*, pages 102–114, 1992.

[121] J. Rouyer. Développement de l'Algorithme d'Unification dans le Calcul des Constructions. Technical Report 1795, INRIA, November 1992.

[122] John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.

[123] A. Saïbi. Axiomatization of a lambda-calculus with explicit-substitutions in the Coq System. Technical Report 2345, INRIA, December 1994.

[124] H. Saidi. Résolution d'équations dans le système t de gödel. Master's thesis, DEA d'Informatique Fondamentale, Université Paris 7, September 1994.

[125] Matthieu Sozeau. Subset coercions in Coq. In *TYPES'06*, volume 4502 of *LNCS*, pages 237–252. Springer, 2007.

[126] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In *TPHOLs'08*, 2008.

[127] T. Streicher. Semantical investigations into intensional type theory, 1993. Habilitationsschrift, LMU Munchen.

[128] Lemme Team. Pcoq a graphical user-interface for Coq. `http://www-sop.inria.fr/lemme/pcoq/`.

[129] The Coq Development Team. The Coq Proof Assistant Reference Manual Version 7.2. Technical Report 255, INRIA, February 2002.

[130] D. Terrasse. Traduction de TYPOL en COQ. Application à Mini ML. Master's thesis, IARFA, September 1992.

[131] L. Théry, Y. Bertot, and G. Kahn. Real theorem provers deserve real user-interfaces. Research Report 1684, INRIA Sophia, May 1992.

[132] 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.

[133] P. Wadler. Efficient compilation of pattern matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[134] P. Weis and X. Leroy. *Le langage Caml*. InterEditions, 1993.

[135] B. Werner. *Une théorie des constructions inductives*. Thèse de doctorat, Université Paris 7, 1994.

Global Index

Tactics Index

Vernacular Commands Index

Index of Error Messages

List of Figures