

The Coq Proof Assistant

Reference Manual

January 22, 2005

Version 8.0 ¹

The Coq Development Team

LogiCal Project

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

V8.0, January 22, 2005

©INRIA 1999-2004 (CoQ versions 7)

©INRIA 2004 (CoQ version 8)

Introduction

This document is the Reference Manual of version 8.0 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 new book [13] on practical uses of the COQ system will be 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 `coqcc` command from the operating system.

These two modes are documented in chapter 12.

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 [111]). These facilities are not documented here. There is also a COQ Integrated Development Environment described in Chapter 14.

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 11.
- In the fourth part more practical tools are documented. First in Chapter 12, the usage of `coq` (batch mode) and `coqtop` (interactive mode) with their options is described. Then, in Chapter 13, various utilities that come with the COQ distribution are presented. Finally, Chapter 14 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, subgoalng 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

G rard 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 Sa bi 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.

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, P. Courtieu, 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 a 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 and Claude Marché.

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.

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

Palaiseau & Orsay, Apr. 2004
Hugo Herbelin & Christine Paulin

Table of contents

I	The language	23
1	The GALLINA specification language	25
1.1	Lexical conventions	25
1.2	Terms	26
1.2.1	Syntax of terms	26
1.2.2	Types	27
1.2.3	Qualified identifiers and simple identifiers	28
1.2.4	Numerals	28
1.2.5	Sorts	28
1.2.6	Binders	29
1.2.7	Abstractions	29
1.2.8	Products	29
1.2.9	Applications	29
1.2.10	Type cast	30
1.2.11	Inferable subterms	30
1.2.12	Local definitions (let-in)	30
1.2.13	Definition by case analysis	30
1.2.14	Recursive functions	30
1.3	The Vernacular	31
1.3.1	Declarations	31
1.3.2	Definitions	33
1.3.3	Inductive definitions	34
1.3.4	Definition of recursive functions	38
1.3.5	Statement and proofs	41
2	Extensions of GALLINA	43
2.1	Record types	43
2.2	Variants and extensions of <code>match</code>	45
2.2.1	Multiple and nested pattern-matching	45
2.2.2	Pattern-matching on boolean values: the <code>if</code> expression	46
2.2.3	Irrefutable patterns: the destructuring <code>let</code>	46
2.2.4	Options for pretty-printing of <code>match</code>	47
2.3	Section mechanism	49
2.3.1	Section <i>ident</i>	49
2.3.2	End <i>ident</i>	49
2.4	Module system	50
2.4.1	Module <i>ident</i>	50
2.4.2	End <i>ident</i>	51
2.4.3	Module <i>ident</i> := <i>module_expression</i>	51

2.4.4	Module Type <i>ident</i>	51
2.4.5	End <i>ident</i>	52
2.4.6	Module Type <i>ident</i> := <i>module_type</i>	52
2.4.7	Declare Module <i>ident</i>	52
2.4.8	End <i>ident</i>	52
2.4.9	Declare Module <i>ident</i> : <i>module_type</i>	52
2.4.10	Import <i>qualid</i>	55
2.4.11	Print Module <i>ident</i>	56
2.4.12	Print Module Type <i>ident</i>	56
2.5	Libraries and qualified names	56
2.5.1	Names of libraries and files	56
2.5.2	Qualified names	57
2.6	Implicit arguments	58
2.6.1	Casual use of implicit arguments	59
2.6.2	Declaration of implicit arguments for a constant	59
2.6.3	Automatic declaration of implicit arguments for a constant	60
2.6.4	Mode for automatic declaration of implicit arguments	61
2.6.5	Controlling strict implicit arguments	61
2.6.6	Controlling contextual implicit arguments	61
2.6.7	Explicit Applications	62
2.6.8	Displaying what the implicit arguments are	62
2.6.9	Explication of implicit arguments for pretty-printing	62
2.6.10	Interaction with subtyping	62
2.6.11	Canonical structures	63
2.6.12	Implicit types of variables	64
2.7	Coercions	64
2.8	Printing constructions in full	65
3	The Coq library	67
3.1	The basic library	67
3.1.1	Notations	67
3.1.2	Logic	67
3.1.3	Datatypes	70
3.1.4	Specification	71
3.1.5	Basic Arithmetics	73
3.1.6	Well-founded recursion	74
3.1.7	Accessing the Type level	75
3.2	The standard library	76
3.2.1	Survey	76
3.2.2	Notations for integer arithmetics	76
3.2.3	Peano's arithmetic (nat)	76
3.2.4	Real numbers library	77
3.2.5	List library	79
3.3	Users' contributions	79
4	Calculus of Inductive Constructions	81
4.1	The terms	81
4.1.1	Sorts	82
4.1.2	Constants	82
4.1.3	Terms	82

Table of contents	15
4.2	Typed terms 83
4.3	Conversion rules 85
4.4	Derived rules for environments 87
4.5	Inductive Definitions 88
4.5.1	Representing an inductive definition 88
4.5.2	Types of inductive objects 90
4.5.3	Well-formed inductive definitions 90
4.5.4	Destructors 91
4.5.5	Fixpoint definitions 96
4.6	Coinductive types 99
4.7	CIC: the Calculus of Inductive Construction with impredicative Set 99
5	The Module System 101
5.1	Modules and module types 101
5.2	Typing Modules 102
II	The proof engine 107
6	Vernacular commands 109
6.1	Displaying 109
6.1.1	Print <i>qualid</i> 109
6.1.2	Print All. 109
6.2	Requests to the environment 109
6.2.1	Check <i>term</i> 109
6.2.2	Eval <i>convtactic</i> in <i>term</i> 110
6.2.3	Extraction <i>term</i> 110
6.2.4	Opaque <i>qualid</i> ₁ . . . <i>qualid</i> _{<i>n</i>} 110
6.2.5	Transparent <i>qualid</i> ₁ . . . <i>qualid</i> _{<i>n</i>} 110
6.2.6	Search <i>qualid</i> 111
6.2.7	SearchAbout <i>qualid</i> 111
6.2.8	SearchPattern <i>term</i> 112
6.2.9	SearchRewrite <i>term</i> 113
6.2.10	Locate <i>qualid</i> 113
6.3	Loading files 113
6.3.1	Load <i>ident</i> 114
6.4	Compiled files 114
6.4.1	Require <i>dirpath</i> 114
6.4.2	Print Modules. 115
6.4.3	Declare ML Module <i>string</i> ₁ .. <i>string</i> _{<i>n</i>} 115
6.4.4	Print ML Modules. 115
6.5	Loadpath 115
6.5.1	Pwd. 115
6.5.2	Cd <i>string</i> 115
6.5.3	Add LoadPath <i>string</i> as <i>dirpath</i> 116
6.5.4	Add Rec LoadPath <i>string</i> as <i>dirpath</i> 116
6.5.5	Remove LoadPath <i>string</i> 116
6.5.6	Print LoadPath. 116
6.5.7	Add ML Path <i>string</i> 116
6.5.8	Add Rec ML Path <i>string</i> 116

6.5.9	Print ML Path <i>string</i> .	116
6.5.10	Locate File <i>string</i> .	117
6.5.11	Locate Library <i>dirpath</i> .	117
6.6	States and Reset	117
6.6.1	Reset <i>ident</i> .	117
6.6.2	Back.	117
6.6.3	Restore State <i>string</i> .	117
6.6.4	Write State <i>string</i> .	118
6.7	Quitting and debugging	118
6.7.1	Quit.	118
6.7.2	Drop.	118
6.7.3	Time <i>command</i> .	118
6.8	Controlling display	118
6.8.1	Set Silent.	118
6.8.2	Unset Silent.	118
6.8.3	Set Printing Width <i>integer</i> .	119
6.8.4	Unset Printing Width.	119
6.8.5	Test Printing Width.	119
6.8.6	Set Printing Depth <i>integer</i> .	119
6.8.7	Unset Printing Depth.	119
6.8.8	Test Printing Depth.	119
7	Proof handling	121
7.1	Switching on/off the proof editing mode	121
7.1.1	Goal <i>form</i> .	121
7.1.2	Qed.	121
7.1.3	Admitted.	122
7.1.4	Theorem <i>ident</i> : <i>form</i> .	122
7.1.5	Proof <i>term</i> .	123
7.1.6	Abort.	123
7.1.7	Suspend.	123
7.1.8	Resume.	124
7.2	Navigation in the proof tree	124
7.2.1	Undo.	124
7.2.2	Set Undo <i>num</i> .	124
7.2.3	Unset Undo.	124
7.2.4	Restart.	124
7.2.5	Focus.	125
7.2.6	Unfocus.	125
7.3	Displaying information	125
7.3.1	Show.	125
7.3.2	Set Hyps Limit <i>num</i> .	126
7.3.3	Unset Hyps Limit.	126
8	Tactics	127
8.1	Invocation of tactics	127
8.2	Explicit proof as a term	127
8.2.1	exact <i>term</i>	127
8.2.2	refine <i>term</i>	128
8.3	Basics	128

8.3.1	assumption	128
8.3.2	clear <i>ident</i>	128
8.3.3	move <i>ident</i> ₁ after <i>ident</i> ₂	129
8.3.4	rename <i>ident</i> ₁ into <i>ident</i> ₂	129
8.3.5	intro	129
8.3.6	apply <i>term</i>	131
8.3.7	set (<i>ident</i> := <i>term</i>)	132
8.3.8	assert (<i>ident</i> : <i>form</i>)	133
8.3.9	generalize <i>term</i>	133
8.3.10	change <i>term</i>	134
8.3.11	Bindings list	134
8.4	Negation and contradiction	135
8.4.1	absurd <i>term</i>	135
8.4.2	contradiction	135
8.5	Conversion tactics	135
8.5.1	cbv <i>flag</i> ₁ ... <i>flag</i> _{<i>n</i>} , lazy <i>flag</i> ₁ ... <i>flag</i> _{<i>n</i>} and compute	135
8.5.2	red	136
8.5.3	hnf	136
8.5.4	simpl	136
8.5.5	unfold <i>qualid</i>	137
8.5.6	fold <i>term</i>	137
8.5.7	pattern <i>term</i>	138
8.5.8	Conversion tactics applied to hypotheses	138
8.6	Introductions	138
8.6.1	constructor <i>num</i>	139
8.7	Eliminations (Induction and Case Analysis)	139
8.7.1	induction <i>term</i>	140
8.7.2	destruct <i>term</i>	142
8.7.3	intros <i>intro_pattern</i> ... <i>intro_pattern</i>	143
8.7.4	double induction <i>ident</i> ₁ <i>ident</i> ₂	144
8.7.5	decompose [<i>qualid</i> ₁ ... <i>qualid</i> _{<i>n</i>}] <i>term</i>	144
8.7.6	functional induction <i>ident term</i> ₁ ... <i>term</i> _{<i>n</i>} .	145
8.8	Equality	145
8.8.1	rewrite <i>term</i>	146
8.8.2	cutrewrite -> <i>term</i> ₁ = <i>term</i> ₂	146
8.8.3	replace <i>term</i> ₁ with <i>term</i> ₂	146
8.8.4	reflexivity	147
8.8.5	symmetry	147
8.8.6	transitivity <i>term</i>	147
8.8.7	subst <i>ident</i>	147
8.8.8	stepl <i>term</i>	147
8.9	Equality and inductive sets	148
8.9.1	decide equality	148
8.9.2	compare <i>term</i> ₁ <i>term</i> ₂	148
8.9.3	discriminate <i>ident</i>	148
8.9.4	injection <i>ident</i>	149
8.9.5	simplify_eq <i>ident</i>	150
8.9.6	dependent rewrite -> <i>ident</i>	150
8.10	Inversion	151

8.10.1	inversion <i>ident</i>	151
8.10.2	Derive Inversion <i>ident</i> with forall($\vec{x}:\vec{T}$), $I \vec{t}$ Sort <i>sort</i>	153
8.10.3	quote <i>ident</i>	154
8.11	Automatizing	154
8.11.1	auto	154
8.11.2	eauto	155
8.11.3	tauto	155
8.11.4	intuition <i>tactic</i>	156
8.11.5	firstorder	156
8.11.6	congruence	157
8.11.7	omega	158
8.11.8	ring <i>term</i> ₁ . . . <i>term</i> _{<i>n</i>}	158
8.11.9	field	158
8.11.10	Add Field	159
8.11.11	fourier	159
8.11.12	autorewrite with <i>ident</i> ₁ . . . <i>ident</i> _{<i>n</i>}	160
8.11.13	Hint Rewrite <i>term</i> ₁ . . . <i>term</i> _{<i>n</i>} : <i>ident</i>	160
8.12	The hints databases for auto and eauto	161
8.12.1	Hint databases defined in the COQ standard library	163
8.12.2	Print Hint	164
8.12.3	Hints and sections	164
8.13	Generation of induction principles with Scheme	164
8.14	Generation of induction principles with Functional Scheme	165
8.15	Simple tactic macros	165
9	The tactic language	167
9.1	Syntax	167
9.2	Semantics	168
9.3	Tactic toplevel definitions	174
10	Detailed examples of tactics	175
10.1	refine	175
10.2	eapply	175
10.3	Scheme	177
10.4	Functional Scheme and functional induction	178
10.5	inversion	180
10.6	autorewrite	183
10.7	quote	184
10.7.1	Introducing variables map	185
10.7.2	Combining variables and constants	186
10.8	Using the tactical language	187
10.8.1	About the cardinality of the set of natural numbers	187
10.8.2	Permutation on closed lists	188
10.8.3	Deciding intuitionistic propositional logic	190
10.8.4	Deciding type isomorphisms	191

Table of contents	19
III User extensions	195
11 Syntax extensions and interpretation scopes	197
11.1 Notations	197
11.1.1 Basic notations	197
11.1.2 Precedences and associativity	198
11.1.3 Complex notations	198
11.1.4 Simple factorization rules	199
11.1.5 Displaying symbolic notations	199
11.1.6 The <code>Infix</code> command	201
11.1.7 Reserving notations	201
11.1.8 Simultaneous definition of terms and notations	201
11.1.9 Displaying informations about notations	201
11.1.10 Locating notations	202
11.1.11 Notations with recursive patterns	203
11.1.12 Notations and binders	203
11.1.13 Summary	204
11.2 Interpretation scopes	204
11.2.1 Global interpretation rules for notations	205
11.2.2 Local interpretation rules for notations	205
11.2.3 The <code>type_ scope</code> interpretation scope	207
11.2.4 Interpretation scopes used in the standard library of COQ	207
11.2.5 Displaying informations about scopes	208
11.3 Abbreviations	208
11.4 Tactic Notations	209
IV Practical tools	211
12 The COQ commands	213
12.1 Interactive use (<code>coqtop</code>)	213
12.2 Batch compilation (<code>coqc</code>)	213
12.3 Resource file	213
12.4 Environment variables	214
12.5 Options	214
13 Utilities	217
13.1 Building a toplevel extended with user tactics	217
13.2 Modules dependencies	218
13.3 Creating a <code>Makefile</code> for COQ modules	218
13.4 Documenting COQ files with <code>coqdoc</code>	218
13.4.1 Principles	219
13.4.2 Usage	221
13.4.3 The <code>coqdoc</code> \LaTeX style file	224
13.5 Exporting COQ theories to XML	224
13.5.1 Practical use of the XML exportation tool	224
13.5.2 Reflection of the logical structure into the file system	225
13.5.3 What is exported?	225
13.5.4 Inner types	225
13.5.5 Interactive exportation commands	226

13.5.6 Applications: rendering, searching and publishing	226
13.5.7 Technical informations	226
13.6 Embedded COQ phrases inside \LaTeX documents	228
13.7 COQ and GNU EMACS	229
13.7.1 The COQ Emacs mode	229
13.7.2 Proof General	229
13.8 Module specification	229
13.9 Man pages	229
14 COQ Integrated Development Environment	231
14.1 Managing files and buffers, basic edition	231
14.2 Interactive navigation into COQ scripts	231
14.3 Try tactics automatically	232
14.4 Vernacular commands, templates	233
14.5 Queries	233
14.6 Compilation	233
14.7 Customizations	234
14.8 Using unicode symbols	234
14.8.1 Displaying unicode symbols	234
14.8.2 Defining an input method for non ASCII symbols	235
14.8.3 Character encoding for saved files	235
14.9 Building a custom COQIDE with user ML code	235
V Addendum to the Reference Manual	237
15 Extended pattern-matching	243
15.1 Patterns	243
15.2 About patterns of parametric types	246
15.3 Matching objects of dependent types	246
15.3.1 Understanding dependencies in patterns	247
15.3.2 When the elimination predicate must be provided	247
15.4 Using pattern matching to write proofs	248
15.5 Pattern-matching on inductive objects involving local definitions	249
15.6 Pattern-matching and coercions	250
15.7 When does the expansion strategy fail ?	250
16 Implicit Coercions	253
16.1 General Presentation	253
16.2 Classes	253
16.3 Coercions	254
16.4 Identity Coercions	254
16.5 Inheritance Graph	254
16.6 Declaration of Coercions	255
16.6.1 Coercion <i>qualid</i> : $class_1 \rightarrow class_2$	255
16.6.2 Identity Coercion <i>ident</i> : $class_1 \rightarrow class_2$	256
16.7 Displaying Available Coercions	257
16.7.1 Print Classes.	257
16.7.2 Print Coercions.	257
16.7.3 Print Graph.	257

Table of contents	21
16.7.4 Print Coercion Paths <i>class₁ class₂</i>	257
16.8 Activating the Printing of Coercions	257
16.8.1 Set Printing Coercions	257
16.8.2 Set Printing Coercion <i>qualid</i>	257
16.9 Classes as Records	257
16.10 Coercions and Sections	258
16.11 Examples	258
17 Omega: a solver of quantifier-free problems in Presburger Arithmetic	263
17.1 Description of omega	263
17.1.1 Arithmetical goals recognized by omega	263
17.1.2 Messages from omega	264
17.2 Using omega	264
17.3 Technical data	265
17.3.1 Overview of the tactic	265
17.3.2 Overview of the <i>OMEGA</i> decision procedure	265
17.4 Bugs	265
18 Extraction of programs in Objective Caml and Haskell	267
18.1 Generating ML code	267
18.2 Extraction options	268
18.2.1 Setting the target language	268
18.2.2 Inlining and optimizations	268
18.2.3 Realizing axioms	269
18.3 Differences between COQ and ML type systems	271
18.4 Some examples	271
18.4.1 A detailed example: Euclidean division	272
18.4.2 Another detailed example: Heapsort	273
18.4.3 The Standard Library	276
18.4.4 Extraction's horror museum	276
18.4.5 Users' Contributions	277
19 The <code>ring</code> tactic	279
19.1 What does this tactic?	279
19.2 The variables map	279
19.3 Is it automatic?	280
19.4 Concrete usage in COQ	280
19.5 Add a ring structure	281
19.6 How does it work?	283
19.7 History of <code>ring</code>	284
19.8 Discussion	284
20 The <code>setoid_replace</code> tactic	287
20.1 Description of <code>setoid_replace</code>	287
20.2 Adding new setoid or morphisms	287
20.3 Adding new morphisms	288
20.4 The tactic itself	288
Bibliography	291

Global Index	298
Tactics Index	306
Vernacular Commands Index	308
Index of Error Messages	311
List of Figures	313

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:

¹This 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 | _
subsequent_letter ::= a..z | A..Z | 0..9 | _ | '
ident ::= first_letter [subsequent_letter...subsequent_letter]

```

All characters are meaningful. In particular, identifiers are case-sensitive. 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 11.

<i>term</i>	::= forall <i>binderlist</i> , <i>term</i>	(1.2.8)
	fun <i>binderlist</i> => <i>term</i>	(1.2.7)
	fix <i>fix_bodies</i>	(1.2.14)
	cofix <i>cofix_bodies</i>	(1.2.14)
	let <i>ident_with_params</i> := <i>term</i> in <i>term</i>	(1.2.12)
	let fix <i>fix_body</i> in <i>term</i>	(1.2.14)
	let cofix <i>cofix_body</i> in <i>term</i>	(1.2.14)
	let ([<i>name</i> , ... , <i>name</i>]) [<i>dep_ret_type</i>] := <i>term</i> in <i>term</i>	(1.2.13, 2.2.1)
	if <i>term</i> [<i>dep_ret_type</i>] then <i>term</i> else <i>term</i>	(1.2.13, 2.2.1)
	<i>term</i> : <i>term</i>	(1.2.10)
	<i>term</i> -> <i>term</i>	(1.2.8)
	<i>term</i> arg ... arg	(1.2.9)
	@ <i>qualid</i> [<i>term</i> ... <i>term</i>]	(2.6.7)
	<i>term</i> % <i>ident</i>	(11.2.2)
	match <i>match_item</i> , ... , <i>match_item</i> [<i>return_type</i>] with	
	[[<i> </i>] <i>equation</i> ... <i>equation</i>] end	(1.2.13)
	<i>qualid</i>	(1.2.3)
	<i>sort</i>	(1.2.5)
	<i>num</i>	(1.2.4)
	-	(1.2.11)
<i>arg</i>	::= <i>term</i>	
	(<i>ident</i> := <i>term</i>)	(2.6.7)
<i>binderlist</i>	::= <i>name</i> ... <i>name</i> [: <i>term</i>]	1.2.6
	<i>binder</i> <i>binderlet</i> ... <i>binderlet</i>	
<i>binder</i>	::= <i>name</i>	1.2.6
	(<i>name</i> ... <i>name</i> : <i>term</i>)	
<i>binderlet</i>	::= <i>binder</i>	1.2.6
	(<i>name</i> [: <i>term</i>] := <i>term</i>)	
<i>name</i>	::= <i>ident</i>	
	-	
<i>qualid</i>	::= <i>ident</i>	
	<i>qualid</i> <i>access_ident</i>	
<i>sort</i>	::= Prop Set Type	

Figure 1.1: Syntax of terms

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*.

<i>ident_with_params</i>	::=	<i>ident</i> [<i>binderlet</i> ... <i>binderlet</i>] [: <i>term</i>]
<i>fix_bodies</i>	::=	<i>fix_body</i> <i>fix_body</i> with <i>fix_body</i> with ... with <i>fix_body</i> for <i>ident</i>
<i>cofix_bodies</i>	::=	<i>cofix_body</i> <i>cofix_body</i> with <i>cofix_body</i> with ... with <i>cofix_body</i> for <i>ident</i>
<i>fix_body</i>	::=	<i>ident binderlet</i> ... <i>binderlet</i> [<i>annotation</i>] [: <i>term</i>] := <i>term</i>
<i>cofix_body</i>	::=	<i>ident_with_params</i> := <i>term</i>
<i>annotation</i>	::=	{ <i>struct ident</i> }
<i>match_item</i>	::=	<i>term</i> [<i>as name</i>] [<i>in term</i>]
<i>dep_ret_type</i>	::=	[<i>as name</i>] <i>return_type</i>
<i>return_type</i>	::=	return <i>term</i>
<i>equation</i>	::=	<i>pattern</i> , ... , <i>pattern</i> => <i>term</i>
<i>pattern</i>	::=	<i>qualid pattern</i> ... <i>pattern</i> <i>pattern as ident</i> <i>pattern</i> % <i>ident</i> <i>qualid</i> — <i>num</i> (<i>pattern</i> , ... , <i>pattern</i>)

Figure 1.2: Syntax of terms (continued)

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 11 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 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.

1.2.6 Binders

Various constructions introduce variables which scope is some of its sub-expressions. There is a uniform syntax for this. A binder may be an (unqualified) identifier: the name to use to refer to this variable. If the variable is not to be used, its name can be `_`. When its type cannot be synthesized by the system, it can be specified with notation `(ident : type)`. There is a notation for several variables sharing the same type: `(ident1...identn : type)`.

Some constructions allow “let-binders”, that is either a binder as defined above, or a variable with a value. The notation is `(ident := term)`. Only one variable can be introduced at the same time. It is also possible to give the type of the variable before the symbol `:=`.

The last kind of binders is the “binder list”. It is either a list of let-binders (the first one not being a variable with value), or `ident1...identn : type` if all variables share the same type.

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*.

1.2.7 Abstractions

The expression “`fun ident : type => term`” denotes the *abstraction* of the variable *ident* of type *type*, over the term *term*. Put in another way, it is function of formal parameter *ident* of type *type* returning *term*.

Keyword `fun` is followed by a “binder list”, so any of the binders of Section 1.2.6 apply. Internally, abstractions are only over one variable. Multiple variable binders are an iteration of the single variable abstraction: notation `fun ident1 ... identn : type => term` stands for `fun ident1 : type => ... fun identn : type => term`. Variables with a value expand 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 it is represented by an iteration of single variable products.

Non dependent product types have a special notation “`A -> B`” stands for “`forall _ : A , B`”. This is to stress on the fact that non dependent product types are usual functional types.

1.2.9 Applications

The expression `term0 term1` denotes the application of term *term₀* to *term₁*.

The expression `term0 term1 ... termn` denotes the application of the term *term₀* to the arguments *term₁* ... then *term_n*. It is equivalent to `... (term0 term1) ... termn`: associativity is to the left.

When using implicit arguments mechanism, implicit positions can be forced a value with notation `(ident := term)` or `(num := term)`. See Section 2.6.7 for details.

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

Since there are redundancies, a term can be type-checked without giving it in totality. Subterms that are left to guess by the type-checker are replaced by “_”.

1.2.12 Local definitions (let-in)

`let ident := term1 in term2` 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 binder1 ... bindern := term1 in term2` stands for `let ident := fun binder1 ... bindern in term2`.

1.2.13 Definition by case analysis

This paragraph only shows simple variants of case analysis. See Section 2.2.1 and Chapter 15 for explanations of the general form.

Objects of inductive types can be deconstructed by a case-analysis construction, also called pattern-matching in functional languages. In its simple form, a case analysis expression is used to analyze the structure of an inductive objects (upon which constructor it is built).

The expression `match term0 return type with pattern1 => term1 | ... | patternn => termn end`, denotes a *pattern-matching* over the term *term*₀ (expected to be of an inductive type *I*). *term*₁...*term*_{*n*} are called branches. In a simple pattern *qualid ident* ... *ident*, the qualified identifier *qualid* is intended to be a constructor. There should be a branch for every constructor of *I*.

The *return_type* is used to compute the resulting type of the whole match expression and the type of the branches. Most of the time, when this type is the same as the types of all the *term*_{*i*}, the annotation is not needed². This annotation has to be given when the resulting type of the whole match depends on the actual *term*₀ matched.

There are specific notations for case analysis on types with one or two constructors: `if / then / else` and `let (...) := ... in ...`. **See also:** section 2.2.1 for details and examples.

See also: Section 2.2.1 for details and examples.

1.2.14 Recursive functions

Expression “`fix ident1 binder1 : type1 := term1 with ... with identn bindern : typen := termn for identi`” 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 identi` is omitted.

The expression “`cofix ident1 binder1 : type1 with ... with identn bindern : typen for identi`” 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 identi` 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.

²except if no equation is given, to match the term in an empty type, e.g. the type `False`

<i>sentence</i>	::=	<i>declaration</i> <i>definition</i> <i>inductive</i> <i>fixpoint</i> <i>statement</i> [<i>proof</i>]
<i>declaration</i>	::=	<i>declaration_keyword</i> <i>assums</i> .
<i>declaration_keyword</i>	::=	Axiom Conjecture Parameter Parameters Variable Variables Hypothesis Hypotheses
<i>assums</i>	::=	<i>ident</i> ... <i>ident</i> : <i>term</i> <i>binder</i> ... <i>binder</i>
<i>definition</i>	::=	Definition <i>ident_with_params</i> := <i>term</i> . Let <i>ident_with_params</i> := <i>term</i> .
<i>inductive</i>	::=	Inductive <i>ind_body</i> with... with <i>ind_body</i> . CoInductive <i>ind_body</i> with... with <i>ind_body</i> .
<i>ind_body</i>	::=	<i>ident</i> [<i>binderlet</i> ... <i>binderlet</i>] : <i>term</i> := [[] <i>ident_with_params</i> ... <i>ident_with_params</i>]
<i>fixpoint</i>	::=	Fixpoint <i>fix_body</i> with... with <i>fix_body</i> . CoFixpoint <i>cofix_body</i> with... with <i>cofix_body</i> .
<i>statement</i>	::=	<i>statement_keyword</i> <i>ident</i> [<i>binderlet</i> ... <i>binderlet</i>] : <i>term</i> .
<i>statement_keyword</i>	::=	Theorem Lemma Definition
<i>proof</i>	::=	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 . . . *ident*_{*n*} : *term* .
Adds *n* parameters with specification *term*
3. Parameter (*ident*_{1,1} . . . *ident*_{1,*k*₁} : *term*₁) . . . (*ident*_{*n*,1} . . . *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 2.3 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 parameterized (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 . . . *ident*_{*n*} : *term* .
Links *term* to names *ident*₁ . . . *ident*_{*n*}.
2. Variable (*ident*_{1,1} . . . *ident*_{1,*k*₁} : *term*₁) . . . (*ident*_{*n*,1} . . . *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 : term1 := term2`.
It checks that the type of *term₂* is definitionally equal to *term₁*, and registers *ident* as being of type *term₁*, and bound to value *term₂*.
2. Definition `ident binder1...bindern : term1 := term2`.
This is equivalent to
Definition `ident : forall binder1...bindern, term1 := fun binder1...bindern => term2`.

Error messages:

1. In environment ... the term: *term₂* does not have type *term₁*.
Actually, it has type *term₃*.

See also: Sections 6.2.4, 6.2.5, 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 : term1 := term2`.

See also: Sections 2.3 (section mechanism), 6.2.4, 6.2.5 (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 :=
  ident1 : type1
| ...
| identn : typen
```

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 : *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 := 0 | S (_:nat).`

In the case where inductive types have no annotations (next section gives an example of such annotations), the positivity condition implies that 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 0
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 0 ->
        (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 the naturals satisfying the clauses *even_0* or *even_SS*. This is why, when we want to prove any predicate *P* over elements of *even*, it is enough to prove it for 0 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*

Parameterized inductive types

Inductive types may be parameterized. Parameters differ from inductive type annotations in the fact that recursive invocations of inductive types must always be done with the same values of parameters as its specification.

The general scheme is:

$$\text{Inductive } \textit{ident} \textit{ binder}_1 \dots \textit{binder}_k : \textit{term} := \textit{ident}_1 : \textit{term}_1 \mid \dots \mid \textit{ident}_n : \textit{term}_n .$$

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 *numth* argument of *ident* must be *ident'* in *type*

See also: Sections 4.5 and 4.

Mutually defined inductive types

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

```
Inductive ident1 : type1 :=
  ident11 : type11
| ...
| identn11 : typen11
with
  ...
with identm : typem :=
  ident1m : type1m
| ...
| identnmm : typenmm.
```

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

It is also possible to parameterize 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 ident1 params : type1 :=
  ident11 : type11
| ..
| identn11 : typen11
with
  ..
with identm params : typem :=
  ident1m : type1m
```

```
| ..
| identnmm : typenmm.
```

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) (f : forest), P f -> P (cons t f)) ->
    forall f1 : forest, P f1
```

Assume we want to parameterize 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 2.3

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 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` (cf. 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 and infinite proof of equality, that is, an infinite object of type $(EqSt\ s_1\ s_2)$. We will see how to introduce infinite objects in Section 1.3.4.

1.3.4 Definition of recursive functions

```
Fixpoint ident params {struct ident0} : type0 := term0
```

This command allows to define inductive objects using a fixed point construction. The meaning of this declaration is to define *ident* a recursive function with arguments specified by $binder_1 \dots binder_n$ 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 $\text{forall } params, type_0$ and the value is equivalent to $\text{fun } params \Rightarrow term_0$.

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 $\{\text{struct } ident\}$ annotation is to let the user tell the system which argument decreases along the recursive calls. This annotation may be left implicit for fixpoints with one argument. For instance, one can define the addition function as :

```
Coq < Fixpoint add (n m:nat) {struct n} : nat :=
Coq <      match n with
Coq <      | 0 => m
Coq <      | S p => S (add p m)
Coq <      end.
add is recursively defined
```

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 0 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 <   | 0 => 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 <   | 0 => 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 <   | 0 => 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 <   | 0 => 0
Coq <   | S p => match p with
Coq <       | 0 => S 0
Coq <       | S q => mod2 q
Coq <   end
Coq <   end.
```

In order to keep the strong normalisation 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*} := *type*_{*m*}
Allows to define simultaneously *ident*₁, ..., *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 8.13.

```
CoFixpoint ident : type0 := term0.
```

The CoFixpoint command 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 0 (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 (cf. 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* .
 It is a synonymous of Theorem
2. Remark *ident* : *type* .
 It is a synonymous of Theorem
3. Fact *ident* : *type* .
 It is a synonymous of Theorem
4. 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 behaviour will be used by further commands. **See also:** 6.2.5 and 8.5.5.

`ProofQed .`

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 6.2.4) 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. `ProofDefined .`
Same as `ProofQed .` but the proof is then declared transparent (see 6.2.5), which means it can be unfolded in conversion tactics (see 8.5).
2. `ProofSave .`
Same as `ProofQed .`
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. `ProofAdmitted .`
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”.

<i>sentence</i>	<code>++=</code>	<i>record</i>
<i>record</i>	<code>::=</code>	<code>Record <i>ident</i> [<i>binderlet</i> ... <i>binderlet</i>] : <i>sort</i> := [<i>ident</i>] { [<i>field</i> ; ... ; <i>field</i>] } .</code>
<i>field</i>	<code>::=</code>	<code><i>name</i> : <i>type</i> <i>name</i> [: <i>term</i>] := <i>term</i></code>

Figure 2.1: Syntax for the definition of `Record`

In the expression

`Record ident params : 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. If *ident*₀ is omitted, the default name `Build_ident` is used. 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 := { ident1 : type1 ; ident2 := term2 ; ident3 : type3 }` 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 :=
  ident0 (ident1:term1) .. (identn:termn).

```

To build an object of type *ident*, one should provide the constructor *ident₀* 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

```

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:

<i>term</i>	++=	<i>term</i> . (<i>qualid</i>)
		<i>term</i> . (<i>qualid</i> <i>arg</i> ... <i>arg</i>)
		<i>term</i> . (@ <i>qualid</i> <i>term</i> ... <i>term</i>)

Figure 2.2: Syntax of Record projections

- (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 16.9 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* *arg₁* ... *arg_n*) to *qualid arg₁* ... *arg_n* *term*, and the syntax *term* . (@*qualid* *term₁* ... *term_n*) to @*qualid term₁* ... *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`.

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 and nested 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 printed under its expanded form.

See also: chapter 15.

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.
Coq <   not is defined
```

can be alternatively written

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

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

$$\text{if } \text{term } [\text{dep_ret_type}] \text{ then } \text{term}_1 \text{ else } \text{term}_2 \equiv \begin{array}{l} \text{match term } [\text{dep_ret_type}] \text{ with} \\ | C_1 \text{ } _ \dots _ \Rightarrow \text{term}_1 \\ | C_2 \text{ } _ \dots _ \Rightarrow \text{term}_2 \\ \text{end} \end{array}$$

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).
Coq <   fun (x : nat) (H : {x = 0} + {x <> 0}) => if H then true else false
Coq <       : 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`

Closed terms (that is not relying on any axiom or variable) in an inductive type having only one constructor, say `foo`, have necessarily the form `(foo ...)`. In this case, the `match` construction can be written with a syntax close to the `let ... in ...` construction. Expression `let (ident1,...,identn) := term0 in term1` performs case analysis on `term0` which must be in an inductive type with one constructor with n arguments. Variables `ident1...identn` are bound to the n arguments of the constructor in expression `term1`. For instance, the definition

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

can be alternatively written

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

Note however that reduction is slightly different from regular `let ... in ...` construction since it can occur only if $term_0$ can be put in constructor form. Otherwise, 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).

The general equivalence for an inductive type with one constructors C is

$$\begin{aligned} & \text{let } (ident_1, \dots, ident_n) [dep_ret_type] := term \text{ in } term' \\ \equiv & \text{match } term [dep_ret_type] \text{ with } C \ ident_1 \dots ident_n \Rightarrow term' \text{ end} \end{aligned}$$

2.2.4 Options for pretty-printing of match

There are three options controlling the pretty-printing of match expressions.

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 synthesisable. 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-synthesise it.

```
Unset Printing Synth.
```

This forces the result type to be always printed.

```
Test Printing Synth.
```

This tells if the non-printing of synthesisable types is on or off. The default is to not print synthesisable 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 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 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 prod.
Cases on elements of prod are printed using a 'let' form

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

2.3 Section mechanism

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

2.3.1 Section *ident*

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

2.3.2 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 generalised 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
```

<i>module_type</i>	<i>::=</i>	<i>ident</i>
		<i>module_type</i> with Definition <i>ident</i> <i>:= term</i>
		<i>module_type</i> with Module <i>ident</i> <i>:= qualid</i>
<i>module_binding</i>	<i>::=</i>	(<i>ident</i> ... <i>ident</i> : <i>module_type</i>)
<i>module_bindings</i>	<i>::=</i>	<i>module_binding</i> ... <i>module_binding</i>
<i>module_expression</i>	<i>::=</i>	<i>qualid</i> ... <i>qualid</i>

Figure 2.3: Syntax of modules

```

Coq < End s1.
x' is discharged.
x'' is discharged.

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 cancelled when the section is closed.

2.4 Module system

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

2.4.1 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*.

2.4.2 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

2.4.3 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*.

2.4.4 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*.

2.4.5 End *ident*

This command closes the interactive module type *ident*.

Error messages:

1. This is not the last opened module type

2.4.6 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*.

2.4.7 Declare Module *ident*

Starts an interactive module declaration. This command is available only in module types.

Variants:

1. Declare Module *ident* *module_bindings*
Starts an interactive declaration of a functor with parameters given by *module_bindings*.
2. Declare Module *ident* *module_bindings* <: *module_type*
Starts an interactive declaration of a functor with parameters given by *module_bindings* (possibly none). The declared output module type is verified against the module type *module_type*.

2.4.8 End *ident*

This command closes the interactive declaration of module *ident*.

2.4.9 Declare Module *ident* : *module_type*

Declares a module of *ident* of type *module_type*. This command is available only in module types.

Variants:

1. Declare Module *ident* *module_bindings* : *module_type*
Declares a functor with parameters *module_bindings* and output module type *module_type*.
2. Declare Module *ident* := *qualid*
Declares a module equal to the module *qualid*.
3. Declare Module *ident* <: *module_type* := *qualid*
Declares a module equal to the module *qualid*, verifying that the module type of the latter is a subtype of *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.
User 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.
P.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 <   Declare Module M2 <: SIG.
Interactive Declaration of Module M2 started

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

```

Coq <      Parameter x : T.
x is assumed

Coq <      End M2.
Module M2 is declared

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`.

2.4.10 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 1102-1103
> Check T.
>      ^
Error: The reference T was not found in the current environment

Coq < Import Mod.

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

```

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* !

2.4.11 Print Module *ident*

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

2.4.12 Print Module Type *ident*

Prints the module type corresponding to *ident*.

2.5 Libraries and qualified names

2.5.1 Names of libraries and files

Libraries The theories developed in COQ are stored in *libraries*. A library is characterised by a name called *root* of the library. The standard library of COQ has root name `Coq` and is known by default when a COQ session starts.

Libraries have a tree structure. E.g., the `Coq` library contains the sub-libraries `Init`, `Logic`, `Arith`, `Lists`, ... The “dot notation” is used to separate the different component of a library name. For instance, the `Arith` library of COQ standard library is written “`Coq.Arith`”.

Remark: no blank is allowed between the dot and the identifier on its right, otherwise the dot is interpreted as the full stop (period) of the command!

Physical paths vs logical paths Libraries and sub-libraries are denoted by *logical directory paths* (written *dirpath* and of which the syntax is the same as *qualid*, see 1.2.3). Logical directory paths can be mapped to physical directories of the operating system using the command (see 6.5.3)

```
Add LoadPath physical_path as dirpath.
```

A library can inherit the tree structure of a physical directory by using the `-R` option to `coqtop` or the command (see 6.5.4)

```
Add Rec LoadPath physical_path as dirpath.
```

Remark: When used interactively with `coqtop` command, COQ opens a library called `Top`.

The file level At some point, (sub-)libraries contain *modules* which coincide with files at the physical level. As for sublibraries, the dot notation is used to denote a specific module of a library. Typically, `Coq.Init.Logic` is the logical path associated to the file `Logic.v` of COQ standard library. Notice that compilation (see 12) is done at the level of files.

If the physical directory where a file `File.v` lies is mapped to the empty logical directory path (which is the default when using the simple form of `Add LoadPath` or `-I` option to `coqtop`), then the name of the module it defines is `File`.

2.5.2 Qualified names

Modules contain constructions (sub-modules, axioms, parameters, definitions, lemmas, theorems, remarks or facts). The (full) name of a construction starts with the logical name of the module in which it is defined followed by the (short) name of the construction. Typically, the full name `Coq.Init.Logic.eq` denotes Leibniz’ equality defined in the module `Logic` in the sublibrary `Init` of the standard library of COQ.

Absolute, partially qualified and short names The full name of a library, module, section, definition, theorem, ... is its *absolute name*. The last identifier (`eq` in the previous example) is its *short name* (or sometimes *base name*). Any suffix of the absolute name is a *partially qualified name* (e.g. `Logic.eq` is a partially qualified name for `Coq.Init.Logic.eq`). Partially qualified names (shortly *qualified name*) are also built from identifiers separated by dots. They are written *qualid* in the documentation.

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).

Visibility COQ maintains a *name table* mapping qualified names to absolute names. This table is modified by the commands `Require` (see 6.4.1), `Import` and `Export` (see 2.4.10) 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 name suffices to denote it. This means that the short or partially qualified name is mapped to the absolute name in COQ name table.

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. Still, 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 (visible as Datatypes.nat)
```

Remark: There is also a name table for sublibraries, modules and sections.

Remark: In versions prior to COQ 7.4, lemmas declared with `Remark` and `Fact` kept in their full name the names of the sections in which they were defined. Since COQ 7.4, they strictly behaves as `Theorem` and `Lemma` do.

See also: Command `Locate` in Section 6.2.10.

Requiring a file A module compiled in a “.vo” file comes with a logical names (e.g. physical file `theories/Init/Datatypes.vo` in the COQ installation directory is bound to the logical module `Coq.Init.Datatypes`). When requiring the file, the mapping between physical directories and logical library should be consistent with the mapping used to compile the file (for modules of the standard library, this is automatic – check it by typing `Print LoadPath`).

The command `Add Rec LoadPath` is also available from `coqtop` and `coqc` by using option `-R`.

2.6 Implicit arguments

An implicit argument of a function is an argument which can be inferred from the knowledge of the type of other arguments of the function, or of the type of the surrounding context of the application. Especially, an implicit argument corresponds to a parameter dependent in the type of the function. Typical implicit arguments are the type arguments in polymorphic functions. More precisely, there are several kinds of 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 opposite, 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 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 non. 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).

2.6.1 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 “`_`”.

2.6.2 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 for all which are the expected implicit arguments of this object. The syntax is

```
Implicit Arguments qualid [ ident ... ident ]
```

where the list of *ident* is the list of parameters to be declared implicit. After this, implicit arguments can just (and have to) be skipped in any expression involving an application of *qualid*.

Example:

```
Coq < Inductive list (A:Set) : Set :=
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
```

Remark: To know which are the implicit arguments of an object, use command `Print Implicit` (see 2.6.8).

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

2.6.3 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 and contextual implicit arguments must be considered or not (see Sections 2.6.5 and 2.6.6).

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
```

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
```

2.6.4 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.

2.6.5 Controlling strict implicit arguments

By default, COQ automatically set implicit only the strict implicit arguments. To relax this constraint, use command

```
Unset Strict Implicit.
```

Conversely, use command `Set Strict Implicit` to restore the strict implicit mode.

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

2.6.6 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.

<i>term</i>	++=	@ <i>qualid</i> <i>term</i> ... <i>term</i>
		@ <i>qualid</i>
		<i>qualid</i> <i>argument</i> ... <i>argument</i>
<i>argument</i>	::=	<i>term</i>
		(<i>ident</i> := <i>term</i>)

Figure 2.4: Syntax for explicitations of implicit arguments

2.6.7 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 . . *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
```

2.6.8 Displaying what the implicit arguments are

To display the implicit arguments associated to an object use command

```
Print Implicit qualid.
```

2.6.9 Explicitation 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.
```

See also: Set Printing All in section 2.8.

2.6.10 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.
```

2.6.11 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_struct* $c_1 \dots c_n$) in the structure *struct* of which the fields are x_1, \dots, 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 :=
  {Carrier :> Set;
   Equal : relation Carrier;
   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 synthesised 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).

2.6.12 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.

2.7 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 $\text{forall } (x_1 : A_1) \dots (x_n : A_n), s$ 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 : class1 >-> class2.
```

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 16.

2.8 Printing constructions in full

Coercions, implicit arguments, the type of pattern-matching, but also notations (see chapter 11) 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.
```


Chapter 3

The CoQ library

The CoQ library is structured into three parts:

The initial library: it contains elementary logical notions and datatypes. 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);

User contributions: Other specification and proof developments coming from the CoQ users' community. These libraries are no longer distributed with the system. They are available by anonymous FTP (see section 3.3).

This chapter briefly reviews these libraries.

3.1 The basic library

This section lists the basic notions and results which are directly available in the standard CoQ system¹.

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 precedence and associativity of very common notations, and avoid users to use them with other precedence, which may be confusing.

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

¹Most 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`, and `Wf`. Module `Logic_Type` also makes it in the initial state

Notation	Precedence	Associativity
$_ \leftrightarrow _$	95	no
$_ \setminus / _$	85	right
$_ /\setminus _$	80	right
$_ \sim _$	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
$_ * _$	40	left
$_ / _$	40	left
$_ - _$	35	right
$_ / _$	35	right
$_ ^ _$	30	right

Figure 3.1: Notations in the initial state

<i>form</i>	::=	True	(True)
		False	(False)
		$\sim form$	(not)
		$form /\setminus form$	(and)
		$form \setminus / form$	(or)
		$form \rightarrow form$	(primitive implication)
		$form \leftrightarrow form$	(iff)
		$\text{forall } ident : type , form$	(primitive for all)
		$\text{exists } ident [: specif] , form$	(ex)
		$\text{exists2 } ident [: specif] , form \& form$	(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 definition² 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.

Programming

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

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

```
Coq < Inductive nat : Set := 0 | 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.
```

²They are in `Datatypes.v`

<i>specif</i>	<code>::=</code>	<i>specif</i> * <i>specif</i>	(prod)
		<i>specif</i> + <i>specif</i>	(sum)
		<i>specif</i> + { <i>specif</i> }	(sumor)
		{ <i>specif</i> } + { <i>specif</i> }	(sumbool)
		{ <i>ident</i> : <i>specif</i> <i>form</i> }	(sig)
		{ <i>ident</i> : <i>specif</i> <i>form</i> & <i>form</i> }	(sig2)
		{ <i>ident</i> : <i>specif</i> & <i>specif</i> }	(sigS)
		{ <i>ident</i> : <i>specif</i> & <i>specif</i> & <i>specif</i> }	(sigS2)
<i>term</i>	<code>::=</code>	(<i>term</i> , <i>term</i>)	(pair)

Figure 3.3: Syntax of datatypes and specifications

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

`identity` is logically equivalent to equality but it lives in sort `Set`. Computationally, it behaves like `unit`.

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.
```

3.1.4 Specification

The following notions³ allows to build new datatypes and specifications. They are available with the syntax shown on Figure 3.3⁴.

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

From such a `(exist x p)` we may in turn extract its witness $x:A$ (using an elimination construct such as `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).
```

³They are defined in module `Specif.v`

⁴This syntax can be found in the module `SpecifSyntax.v`

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

```
Coq < Inductive sigS (A:Set) (P:A -> Set) : Set := existS (x:A) ( _:P x).
Coq < Section sigSprojections.
Coq < Variable A : Set.
Coq < Variable P : A -> Set.
Coq < Definition projS1 (H:sigS A P) := let (x, h) := H in x.
Coq < Definition projS2 (H:sigS A P) :=
Coq <   match H return P (projS1 H) with
Coq <     existS x h => h
Coq <   end.
Coq < End sigSprojections.
Coq < Inductive sigS2 (A: Set) (P Q:A -> Set) : Set :=
Coq <   existS2 (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 $A:\text{Set}$ and $B:\text{Prop}$ 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 constructs builds a sum between a data type $A:\text{Set}$ 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` and `Prop` in a way which is consistent with the realizability interpretation.

```
Coq < Definition except := False_rec.
Coq < Notation Except := (except _).
Coq < Theorem absurd_set : forall (A:Prop) (C:Set), A -> ~ A -> C.
Coq < Theorem and_rec :
Coq < forall (A B:Prop) (P:Set), (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 multiplication⁵. 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.

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.
```

⁵This is in module `Peano.v`

```

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

Coq < Lemma plus_n_0 : forall n:nat, n = plus n 0.
Coq < Lemma plus_n_Sm : forall n m:nat, S (plus n m) = plus n (S m).

Coq < Fixpoint mult (n m:nat) {struct n} : nat :=
Coq <   match n with
Coq <   | 0 => 0
Coq <   | S p => m + mult p m
Coq <   end.

Coq < Lemma mult_n_0 : forall n:nat, 0 = mult n 0.
Coq < Lemma mult_n_Sm : forall n m:nat, plus (mult n m) n = mult 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, le n m -> le n (S m).
Coq < Infix "+" := plus : 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 induction⁶.

```

Coq < Section Well_founded.
Coq < Variable A : Set.
Coq < Variable R : A -> A -> Prop.
Coq < Inductive Acc : A -> Prop :=
Coq <   Acc_intro : forall x:A, (forall y:A, R y x -> Acc y) -> Acc x.
Coq < Lemma Acc_inv : forall x:A, Acc x -> forall y:A, R y x -> Acc y.

```

⁶This is defined in module `Wf.v`

```

Coq < Section AccRec.
Coq < Variable P : A -> Set.
Coq < Variable F :
Coq <   forall x:A,
Coq <   (forall y:A, R y x -> Acc y) -> (forall y:A, R y x -> P y) -> P x.
Coq < Fixpoint Acc_rec (x:A) (a:Acc x) {struct a} : P x :=
Coq <   F x (Acc_inv x a)
Coq <   (fun (y:A) (h:R y x) => Acc_rec y (Acc_inv x a y h)).
Coq < End AccRec.

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.

```

`Acc_rec` 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 -> Set.
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.

```

3.1.7 Accessing the Type level

The basic library includes the definitions⁷ of the counterparts of some datatypes 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 datatypes at the Type level.

⁷This is in module `Logic_Type.v`

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
ZAarith	Basic integer arithmetic
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.)
IntMap	Representation of finite sets by an efficient structure of map (trees indexed by binary integers).
Reals	Axiomatization of Real Numbers (classical, basic functions, integer part, fractional part, limit, derivative, Cauchy series, power series and results,... Requires the ZAarith library).
Relations	Relations (definitions and basic results).
Sorting	Sorted list (basic definitions and heapsort correctness).
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.

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 `ZAarith` 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 ZAarith.
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`.

⁸<http://coq.inria.fr>

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

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

Notation	Interpretation
$_ < _$	lt
$x \leq y$	le
$_ > _$	gt
$x \geq y$	ge
$x < y < z$	$x < y \wedge y < z$
$x < y \leq z$	$x < y \wedge y \leq z$
$x \leq y < z$	$x \leq y \wedge y < z$
$x \leq y \leq z$	$x \leq y \wedge y \leq z$
$_ + _$	plus
$_ - _$	minus
$_ * _$	mult

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

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

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

`: 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 conjunctions.

```
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
```

Notation	Interpretation	Precedence	Associativity
<code>_ ++ _</code>	<code>app</code>	60	right
<code>_ :: _</code>	<code>cons</code>	60	right

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

- `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>.

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:

<code>length</code>	<code>length</code>
<code>head</code>	first element (with default)
<code>tail</code>	all but first element
<code>app</code>	concatenation
<code>rev</code>	reverse
<code>nth</code>	accessing n -th element (with default)
<code>map</code>	applying a function
<code>flat_map</code>	applying a function returning lists
<code>fold_left</code>	iterator (from head to tail)
<code>fold_right</code>	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 [61] provides an introduction to inductive and coinductive definitions in Coq. In their book [13], Bertot and Castéran give a precise description of the pCIC based on numerous practical examples. Barras [9], Werner [118] and Paulin-Mohring [104] are the most recent theses dealing with Inductive Definitions. Coquand-Huet [27, 28, 29] introduces the Calculus of Constructions. Coquand-Paulin [30] 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 => (mult x x)` (assuming `mult` is already defined) but may build also predicates over the natural numbers. For instance `fun x : nat => (x = x)` will represent a predicate P , informally written in mathematics $P(x) \equiv x = x$. If P has

type $\text{nat} \rightarrow \text{Prop}$, $(P\ x)$ is a proposition, furthermore forall $x : \text{nat}$, $(P\ x)$ 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 $\text{Type}(i)$ for any integer i . We call S the set of sorts which is defined by:

$$S \equiv \{\text{Prop}, \text{Set}, \text{Type}(i) \mid i \in \mathbb{N}\}$$

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

The user will never mention explicitly the index i when referring to the universe $\text{Type}(i)$. 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 $\text{Type} : \text{Type}$.

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

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 $(t t_1 \dots t_n)$ represents $(\dots (t t_1) \dots t_n)$. 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 $\text{Assum}(\Gamma)(c : T)$ 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 $\text{Def}(\Gamma)(c := t : T)$ 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.

W-E

$$\mathcal{WF}(\emptyset)(\emptyset)$$

W-S

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

Def

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

Ax

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

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

Var

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

¹This requirement could be relaxed if we instead introduced an explicit mechanism for instantiating constants. At the external level, the Coq engine works accordingly to this view that all the definitions in the environment were built in a sub-context of the current context.

Const

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

Prod

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

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

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

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

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

Remark: We may have $\mathbf{let} \ x := t \ \mathbf{in} \ u$ 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 [21].

ι -reduction. A specific conversion rule is associated to the inductive objects in the environment. We shall give later on (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 [103, 118].

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.

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

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.

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

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 $E[\Gamma] \vdash t_1 \triangleright \dots \triangleright u$ and $E[\Gamma] \vdash t_2 \triangleright \dots \triangleright u$. 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 $E[\Gamma] \vdash \text{Type}(i) \leq_{\beta\delta\iota\zeta} \text{Type}(j)$,
3. for any i , $E[\Gamma] \vdash \text{Prop} \leq_{\beta\delta\iota\zeta} \text{Type}(i)$,
4. for any i , $E[\Gamma] \vdash \text{Set} \leq_{\beta\delta\iota\zeta} \text{Type}(i)$,
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 $\lambda x_1 : T_1, \dots \lambda x_k : T_k, (t_0 t_1 \dots t_n)$ 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:

$$\lambda x_1 : T_1, \dots \lambda x_k : T_k, (\lambda x : T, u_0 t_1 \dots t_n) \triangleright \lambda (x_1 : T_1) \dots (x_k : T_k), (u_0 \{x/t_1\} t_2 \dots t_n)$$

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 :

$$t \triangleright \dots \triangleright \lambda x_1 : T_1, \dots \lambda x_k : T_k, (v u_1 \dots u_m)$$

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:

$$\frac{\mathcal{WF}(E; \mathbf{Def}(\Gamma)(c := t : T); F)[\Delta]}{\mathcal{WF}(E; F\{c/t\})[\Delta\{c/t\}]}$$

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:

$$\frac{\mathcal{WF}(E; \mathbf{Def}(\Gamma :: (x : U))(c := t : T); F)[\Delta] \quad \mathcal{WF}(E)[\Gamma]}{\mathcal{WF}(E; \mathbf{Def}(\Gamma)(c := \lambda x : U, t : \forall x : U, T); F\{c/(c x)\})[\Delta\{c/(c x)\}]}$$

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:

$$\frac{\mathcal{WF}(E; \text{Def}(\Delta)(c := t : T))[\Gamma]}{\mathcal{WF}(E; \text{Def}(\Delta)(c := t : T))[\Delta]}$$

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 precisising 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 $\text{Ind}(\Gamma)(\Gamma_I := \Gamma_C)$ 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:

$$\text{Ind}()(\text{nat} : \text{Set} := \text{O} : \text{nat}, \text{S} : \text{nat} \rightarrow \text{nat})$$

In a context with a variable $A : \text{Set}$, the lists of elements in A is represented by:

$$\text{Ind}(A : \text{Set})(\text{List} : \text{Set} := \text{nil} : \text{List}, \text{cons} : A \rightarrow \text{List} \rightarrow \text{List})$$

Assuming Γ_I is $[I_1 : A_1; \dots; I_k : A_k]$, and Γ_C is $[c_1 : C_1; \dots; c_n : C_n]$, the general typing rules are:

$$\frac{\text{Ind}(\Gamma)(\Gamma_I := \Gamma_C) \in E \quad j = 1 \dots k}{(I_j : A_j) \in E}$$

$$\frac{\text{Ind}(\Gamma)(\Gamma_I := \Gamma_C) \in E \quad i = 1..n}{(c_i : C_i) \in E}$$

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`. As they were defined above, the type `List` can only be used in an environment where we have a variable $A : \text{Set}$. 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 $\text{Set} \rightarrow \text{Set}$:

$$\text{Ind}()(\text{List} : \text{Set} \rightarrow \text{Set} := \text{nil} : (A : \text{Set})(\text{List } A), \text{cons} : (A : \text{Set})A \rightarrow (\text{List } A) \rightarrow (\text{List } A))$$

There are drawbacks to this point of view. The information which says that (List nat) is an inductively defined **Set** has been lost.

In the system, we keep track in the syntax of the context of parameters. The idea of these parameters is that they can be instantiated and still we have an inductive definition for which we know the specification.

Formally the representation of an inductive declaration will be $\text{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C)$ for an inductive definition valid in a context Γ with parameters Γ_P , a context of definitions Γ_I and a context of constructors Γ_C . The occurrences of the variables of Γ_P in the contexts Γ_I and Γ_C are bound.

The definition $\text{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C)$ will be well-formed exactly when $\text{Ind}(\Gamma, \Gamma_P)(\Gamma_I := \Gamma_C)$ is. If Γ_P is $[p_1 : P_1; \dots; p_r : P_r]$, an object in $\text{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C)$ applied to q_1, \dots, q_r will behave as the corresponding object of $\text{Ind}(\Gamma)(\Gamma_I\{(p_i/q_i)_{i=1..r}\} := \Gamma_C\{(p_i/q_i)_{i=1..r}\})$.

Examples The declaration for parameterized lists is:

$$\text{Ind}()[A : \text{Set}](\text{List} : \text{Set} := \text{nil} : \text{List}, \text{cons} : A \rightarrow \text{List} \rightarrow \text{List})$$

The declaration for the length of lists is:

$$\begin{aligned} \text{Ind}()[A : \text{Set}](\text{Length} : (\text{List } A) \rightarrow \text{nat} \rightarrow \text{Prop} := & \text{Lnil} : (\text{Length } (\text{nil } A) \text{ O}), \\ \text{Lcons} : \forall a : A, \forall l : (\text{List } A), \forall n : \text{nat}, & (\text{Length } l \text{ } n) \rightarrow (\text{Length } (\text{cons } A \text{ } a \text{ } l) \text{ } (\text{S } n))) \end{aligned}$$

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

$$\begin{aligned} \text{Ind}()(\text{tree} : \text{Set}, \text{forest} : \text{Set} := \\ \text{node} : \text{forest} \rightarrow \text{tree}, \text{emptyf} : \text{forest}, \text{consf} : \text{tree} \rightarrow \text{forest} \rightarrow \text{forest}) \end{aligned}$$

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 inductive declaration in COQ is slightly different from the one we described theoretically. The difference is that in the type of constructors the inductive definition is explicitly applied to the parameters variables. The COQ type-checker verifies that all parameters are applied in the correct manner in each recursive call. In particular, the following definition will not be accepted because there is an occurrence of **List** which is not applied to the parameter variable:

```
Coq < Inductive list' (A:Set) : Set :=
Coq <   | nil' : list' A
Coq <   | cons' : A -> list' (A -> A) -> list' A.
Coq < Coq < Error: The 1st argument of "list'" must be "A" in
A -> list' (A -> A) -> list' A
```

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_P is $[p_1 : P_1; \dots; p_r : P_r]$, Γ_I is $[I_1 : A_1; \dots; I_k : A_k]$, and Γ_C is $[c_1 : C_1; \dots; c_n : C_n]$,

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

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

Example. We have $(\text{List} : \text{Set} \rightarrow \text{Set})$, $(\text{cons} : \forall A : \text{Set}, A \rightarrow (\text{List } A) \rightarrow (\text{List } A))$, $(\text{Length} : \forall A : \text{Set}, (\text{List } A) \rightarrow \text{nat} \rightarrow \text{Prop})$, $\text{tree} : \text{Set}$ and $\text{forest} : \text{Set}$.

From now on, we write List_A instead of $(\text{List } A)$ and Length_A for $(\text{Length } A)$.

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 $A \rightarrow \text{Set}$ or $\forall A : \text{Prop}, A \rightarrow \text{Prop}$ are arities of sort respectively **Set** and **Prop**). A *type of constructor of I* is either a term $(I \ t_1 \dots t_n)$ or $\forall x : T, C$ with C 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:

- $T = (X \ t_1 \dots t_n)$ 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 $(X \ t_1 \dots t_n)$ 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 $(I \ a_1 \dots a_m \ t_1 \dots t_p)$ where I is the name of an inductive declaration of the form $\text{Ind}(\Gamma)[p_1 : P_1; \dots; p_m : P_m](I : A := c_1 : C_1; \dots; c_n : C_n)$ (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 types of constructor $C_i\{p_j / a_j\}_{j=1 \dots m}$ of I satisfy the imbricated positivity condition for X

The type of constructor T of I *satisfies the imbricated positivity condition* for a constant X in the following cases:

- $T = (I \ t_1 \dots t_n)$ and X does not occur in any t_i
- $T = \forall x : U, V$ and X occurs only strictly positively in U and the type V satisfies the imbricated positivity condition for X

Example X occurs strictly positively in $A \rightarrow X$ or $X * A$ or $(\text{list } X)$ but not in $X \rightarrow A$ or $(X \rightarrow A) \rightarrow A$ nor $(\text{neg } A)$ assuming the notion of product and lists were already defined and neg is an inductive definition with declaration $\text{Ind}()[A : \text{Set}](\text{neg} : \text{Set} := \text{neg} : (A \rightarrow \text{False}) \rightarrow \text{neg})$. Assuming X has arity $\text{nat} \rightarrow \text{Prop}$ and ex is the inductively defined existential quantifier, the occurrence of X in $(\text{ex } \text{nat } \lambda n : \text{nat}, (X \ n))$ 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 $[I_1 : A_1; \dots; I_k : A_k]$ and Γ_C is $[c_1 : C_1; \dots; c_n : C_n]$.

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

providing the following side conditions hold:

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

One can remark that there is a constraint between the sort of the arity of the inductive type and the sort of the type of its constructors which will always be satisfied for the impredicative sort (**Prop**) but may fail to define inductive definition on sort **Set** and generate constraints between universes for inductive definitions in types.

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 < User 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 $(\text{Type}_i \rightarrow \text{Prop}) \rightarrow \text{Type}_j$ with the constraint $i < j$.

```
Coq < Inductive exType (P:Type->Prop) : Type
Coq <   := exT_intro : forall X:Type, P X -> exType P.
```

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 normalising reduction, we cannot accept any sort of recursion (even terminating). So the basic idea is to restrict ourselves to primitive recursive functions and functionals.

For instance, assuming a parameter $A : \mathbf{Set}$ exists in the context, we want to build a function `length` of type $\text{List_A} \rightarrow \mathbf{nat}$ which computes the length of the list, so such that $(\text{length nil}) = \mathbf{O}$ and $(\text{length (cons } A \ a \ l)) = (\mathbf{S} (\text{length } l))$. 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 $\forall l : \text{List_A}, (\text{Length_A } l \ (\text{length } l))$ it is enough to prove: $(\text{Length_A nil } (\text{length nil}))$ and

$$\forall a : A, \forall l : \text{List_A}, (\text{Length_A } l \ (\text{length } l)) \rightarrow (\text{Length_A (cons } A \ a \ l) \ (\text{length (cons } A \ a \ l))).$$

which given the conversion equalities satisfied by `length` is the same as proving: $(\text{Length_A nil } \mathbf{O})$ and $\forall a : A, \forall l : \text{List_A}, (\text{Length_A } l \ (\text{length } l)) \rightarrow (\text{Length_A (cons } A \ a \ l) \ (\mathbf{S} (\text{length } l)))$.

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 [25]. 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 $m = (c_i \ u_1 \dots u_{p_i})$ for each constructor of I .

The COQ term for this proof will be written :

$$\text{match } m \text{ with } (c_1 \ x_{11} \dots x_{1p_1}) \Rightarrow f_1 \mid \dots \mid (c_n \ x_{n1} \dots x_{np_n}) \Rightarrow f_n \text{ end}$$

In this expression, if m is a term built from a constructor $(c_i \ u_1 \dots u_{p_i})$ then the expression will behave as it is specified with i -th branch and will reduce to f_i where the $x_{i1} \dots x_{ip_i}$ are replaced by the $u_1 \dots u_p$ 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. COQ can sometimes infer this predicate but sometimes not. The concrete syntax for describing this predicate uses the `as...return` construction. The predicate will be explicited using the syntax :

$$\text{match } m \text{ as } x \text{ return } (P \ x) \text{ with } (c_1 \ x_{11} \dots x_{1p_1}) \Rightarrow f_1 \mid \dots \mid (c_n \ x_{n1} \dots x_{np_n}) \Rightarrow f_n \text{ end}$$

For the purpose of presenting the inference rules, we use a more compact notation :

$$\text{case}(m, (\lambda x, P), \lambda x_{11} \dots x_{1p_1}, f_1 \mid \dots \mid \lambda x_{n1} \dots x_{np_n}, f_n)$$

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

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

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

$$\lambda l : \text{List_A}, \text{case}(l, \text{bool}, \text{nil} \Rightarrow \text{true} \mid (\text{cons } a \ m) \Rightarrow \text{false})$$

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

The 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 $[I : A \mid B]$ is defined as the smallest relation satisfying the following rules: We write $[I \mid B]$ for $[I : A \mid B]$ where A is the type of I .

Prod

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

Set& Type

$$\frac{s_1 \in \{\text{Set}, \text{Type}(j)\}, s_2 \in \mathcal{S}}{[I : s_1 \mid I \rightarrow s_2]}$$

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

$$[I : \text{Prop} \mid I \rightarrow \text{Prop}]$$

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 $(\text{or } A \ B)$ in this term is needed for computing the boolean value.

In general, if I has type **Prop** then P cannot have type $I \rightarrow \text{Set}$, 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 $I \rightarrow \text{Type}$ cannot be allowed because it trivially implies the elimination on P of type $I \rightarrow \text{Set}$ 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 $I \rightarrow \text{Type}$ 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

$$\frac{I \text{ is an empty or singleton definition } \quad s \in \mathcal{S}}{[I : \text{Prop} | I \rightarrow s]}$$

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 $\text{List_A} \rightarrow s$ for $s \in \mathcal{S}$.

$\{(\text{cons } A)\}^P \equiv \forall a : A, \forall l : \text{List_A}, (P (\text{cons } A a l))$.

For `Length_A`, the type of P will be $\forall l : \text{List_A}, \forall n : \text{nat}, (\text{Length_A } l n) \rightarrow \text{Prop}$ and the expression $\{(\text{Lcons } A)\}^P$ is defined as:

$\forall a : A, \forall l : \text{List_A}, \forall n : \text{nat}, \forall h : (\text{Length_A } l n), (P (\text{cons } A a l) (\text{S } n) (\text{Lcons } A a l n h)))$.

If P does not depend on its third argument, we find the more natural expression:

$\forall a : A, \forall l : \text{List_A}, \forall n : \text{nat}, (\text{Length_A } l n) \rightarrow (P (\text{cons } A a l) (\text{S } n))$.

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

match

$$\frac{E[\Gamma] \vdash c : (I q_1 \dots q_r t_1 \dots t_s) \quad E[\Gamma] \vdash P : B \quad [(I q_1 \dots q_r) | B] \quad (E[\Gamma] \vdash f_i : \{(c_{p_i} q_1 \dots q_r)\}^P)_{i=1 \dots l}}{E[\Gamma] \vdash \text{case}(c, P, f_1 \dots f_l) : (P t_1 \dots t_s c)}$$

provided I is an inductive type in a declaration $\text{Ind}(\Delta)[\Gamma_P](\Gamma_I := \Gamma_C)$ with $|\Gamma_P| = r$, $\Gamma_C = [c_1 : C_1; \dots; c_n : C_n]$ and $c_{p_1} \dots c_{p_l}$ 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).

$$\frac{l : \text{List_A} \quad P : \text{List_A} \rightarrow s \quad f_1 : (P (\text{nil } A)) \quad f_2 : \forall a : A, \forall l : \text{List_A}, (P (\text{cons } A a l))}{\text{case}(l, P, f_1 f_2) : (P l)}$$

$$\frac{\begin{array}{c} H : (\text{Length_A } L N) \\ P : \forall l : \text{List_A}, \forall n : \text{nat}, (\text{Length_A } l n) \rightarrow \text{Prop} \\ f_1 : (P (\text{nil } A) \text{O } \text{Lnil}) \\ f_2 : \forall a : A, \forall l : \text{List_A}, \forall n : \text{nat}, \forall h : (\text{Length_A } l n), (P (\text{cons } A a n) (\text{S } n) (\text{Lcons } A a l n h))) \end{array}}{\text{case}(H, P, f_1 f_2) : (P L N H)}$$

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:

$$\text{case}((c_{p_i} q_1 \dots q_r a_1 \dots a_m), P, f_1 \dots f_l)$$

with c_{p_i} the i -th constructor of the inductive type I with r parameters.

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

$$\text{case}((c_{p_i} q_1 \dots q_r a_1 \dots a_m), P, f_1 \dots f_n) \triangleright_{\iota} (f_i a_1 \dots a_m)$$

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) :

$$\text{fix } f_1(\Gamma_1) : A_1 := t_1 \text{ with } \dots \text{ with } f_n(\Gamma_n) : A_n := t_n$$

The terms are obtained by projections from this set of declarations and are written

$$\text{fix } f_1(\Gamma_1) : A_1 := t_1 \text{ with } \dots \text{ with } f_n(\Gamma_n) : A_n := t_n \text{ for } f_i$$

In the inference rules, we represent such a term by

$$\text{Fix } f_i \{ f_1 : A'_1 := t'_1 \dots f_n : A'_n := t'_n \}$$

with t'_i (resp. A'_i) representing the term t_i abstracted (resp. generalised) 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

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

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

$$\forall P : \text{nat} \rightarrow \text{Prop}, (P \text{ O}) \rightarrow ((n : \text{nat}) (P n) \rightarrow (P (\text{S } n))) \rightarrow \forall n : \text{nat}, (P n)$$

can be represented by the term:

$$\lambda P : \text{nat} \rightarrow \text{Prop}, \lambda f : (P \text{ O}), \lambda g : (\forall n : \text{nat}, (P n) \rightarrow (P (\text{S } n))), \\ \text{Fix } h \{ h : \forall n : \text{nat}, (P n) := \lambda n : \text{nat}, \text{case}(n, P, f \lambda p : \text{nat}, (g p (h p))) \}$$

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 [59].

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

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

where k_i are positive integers. Each A_i should be a type (reducible to a term) starting with at least k_i products $\forall y_1 : B_1, \dots \forall y_{k_i} : B_{k_i}, A'_i$ 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 $\text{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C)$, the type of a constructor c have the form $\forall p_1 : P_1, \dots, \forall p_r : P_r, \forall x_1 : T_1, \dots, \forall x_r : T_r, (I_j p_1 \dots p_r t_1 \dots t_s)$ the recursive arguments will correspond to T_i in which one of the I_l occurs.

The main rules for being structurally smaller are the following:

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

- $(t \ u), \lambda x : u, t$ when t is structurally smaller than y .
- **case** $(c, P, f_1 \dots f_n)$ 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 $C_q \equiv \forall y_1 : B_1, \dots, \forall y_k : B_k, (I a_1 \dots a_k)$ and can consequently be written $\lambda y_1 : B'_1, \dots, \lambda y_k : B'_k, g_i$. (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 <   | 0 => m
Coq <   | S p => S (plus p m)
Coq <   end.
plus is recursively defined

Coq < Print plus.
plus =
fix plus (n m : nat) {struct n} : nat :=
  match n with
  | 0 => 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 => 0
Coq <   | cons a l' => S (lgth A l')
Coq <   end.
lgth is recursively defined

Coq < Print lgth.
lgth =
fix lgth (A : Set) (l : list A) {struct l} : nat :=
  match l with
  | nil => 0
  | 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 => 0
Coq < | consf t f => plus (sized t) (sized f)
Coq < end.
sized, sizedf are recursively defined

Coq < Print sized.
sized =
fix sized (t : tree) : nat :=
  let (f) := t in S (sizedf f)
with sizedf (f : forest) : nat :=
  match f with
  | emptyf => 0
  | consf t f0 => plus (sized t) (sizedf f0)
end
for sized
  : 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:

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

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

We can illustrate this behavior on examples.

```

Coq < Goal 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, sized (node f) = S (sizedf f).
1 subgoal

=====
forall f : forest, sized (node f) = S (sizedf 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 2398-2409
> 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 8.13.

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 [60, 61].

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 2658-2668
> 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

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

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

$$\frac{s \in \{\mathbf{Prop}, \mathbf{Set}\}}{[I : \mathbf{Set} | I \rightarrow s]} \quad \frac{I \text{ is a small inductive definition} \quad s \in \{\mathbf{Type}(i)\}}{[I : \mathbf{Set} | I \rightarrow s]}$$

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 $Impl$ and is either a definition of a constant, an assumption, a definition of an inductive or a definition of a module or a module type abbreviation.

Module expression. It is denoted by M and can be:

- an access path p
- a structure `Struct $Impl$;...; $Impl$ End`
- a functor `Functor($X : T$) M'` , where X is a module variable, T is a module type and M' is a module expression
- an application of access paths $p'p''$

Signature element. It is denoted by $Spec$, it is a specification of a constant, an assumption, an inductive, a module or a module type abbreviation.

Module type, denoted by T can be:

- a module type name
- an access path p
- a signature `Sig $Spec$;...; $Spec$ End`
- a functor type `Funsig($X : T'$) T''` , where T' and T'' are module types

Module definition, written `Mod($X : T := M$)` can be a structure element. It consists of a module variable X , a module type T and a module expression M .

Module specification, written $\text{ModS}(X : T)$ or $\text{ModSEq}(X : T == p)$ can be a signature element or a part of an environment. It consists of a module variable X , a module type T and, optionally, a module path p .

Module type abbreviation, written $\text{ModType}(S := T)$, where S is a module type name and T is a module type.

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 signature elements. Terms, apart from variables, constants and complex terms, include also access paths.

We also need additional typing judgments:

- $E[] \vdash \mathcal{WF}(T)$, denoting that a module type T is well-formed,
- $E[] \vdash M : T$, denoting that a module expression M has type T in environment E .
- $E[] \vdash \text{Impl} : \text{Spec}$, denoting that an implementation Impl verifies a specification Spec
- $E[] \vdash T_1 <: T_2$, denoting that a module type T_1 is a subtype of a module type T_2 .
- $E[] \vdash \text{Spec}_1 <: \text{Spec}_2$, denoting that a specification Spec_1 is more precise than a specification Spec_2 .

The rules for forming module types are the following:

WF-SIG

$$\frac{\mathcal{WF}(E; E')[]}{E[] \vdash \mathcal{WF}(\text{Sig } E' \text{ End})}$$

WF-FUN

$$\frac{E; \text{ModS}(X : T)[] \vdash \mathcal{WF}(T')}{E[] \vdash \mathcal{WF}(\text{Funsig}(X : T) T')}$$

Rules for typing module expressions:

MT-STRUCT

$$\frac{\begin{array}{c} E[] \vdash \mathcal{WF}(\text{Sig } \text{Spec}_1; \dots; \text{Spec}_n \text{ End}) \\ E; \text{Spec}_1; \dots; \text{Spec}_{i-1}[] \vdash \text{Impl}_i : \text{Spec}_i \text{ for } i = 1 \dots n \end{array}}{E[] \vdash \text{Struct } \text{Impl}_1; \dots; \text{Impl}_n \text{ End} : \text{Sig } \text{Spec}_1; \dots; \text{Spec}_n \text{ End}}$$

MT-FUN

$$\frac{E; \text{ModS}(X : T)[] \vdash M : T'}{E[] \vdash \text{Functor}(X : T) M : \text{Funsig}(X : T) T'}$$

MT-APP

$$\frac{\begin{array}{c} E[] \vdash p : \text{Funsig}(X_1 : T_1) \dots \text{Funsig}(X_n : T_n) T' \\ E[] \vdash p_i : T_i\{X_1/p_1 \dots X_{i-1}/p_{i-1}\} \text{ for } i = 1 \dots n \end{array}}{E[] \vdash p p_1 \dots p_n : T'\{X_1/p_1 \dots X_n/p_n\}}$$

MT-SUB

$$\frac{E[] \vdash M : T \quad E[] \vdash T <: T'}{E[] \vdash M : T'}$$

MT-STR

$$\frac{E[] \vdash p : T}{E[] \vdash p : T/p}$$

The last rule, called strengthening is used to make all module fields manifestly equal to themselves. The notation T/p has the following meaning:

- if $T = \text{Sig } \text{Spec}_1; \dots; \text{Spec}_n \text{ End}$ then $T/p = \text{Sig } \text{Spec}_1/p; \dots; \text{Spec}_n/p \text{ End}$ where Spec/p is defined as follows:
 - $\text{Def}()(c := U : t)/p = \text{Def}()(c := U : t)$
 - $\text{Assum}()(c : U)/p = \text{Def}()(c := p.c : U)$
 - $\text{ModS}(X : T)/p = \text{ModSEq}(X : T/p.X == p.X)$
 - $\text{ModSEq}(X : T == p')/p = \text{ModSEq}(X : T/p == p')$
 - $\text{Ind}()[\Gamma_P](\Gamma_C := \Gamma_I)/p = \text{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I)$
 - $\text{Ind}_{p'}()[\Gamma_P](\Gamma_C := \Gamma_I) /p = \text{Ind}_{p'}()[\Gamma_P](\Gamma_C := \Gamma_I)$
- if $T = \text{Funsig}(X : T') T''$ then $T/p = T$
- if T is an access path or a module type name, then we have to unfold its definition and proceed according to the rules above.

The notation $\text{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I)$ denotes an inductive definition that is definitionally equal to the inductive definition in the module denoted by the path p . All rules which have $\text{Ind}()[\Gamma_P](\Gamma_C := \Gamma_I)$ as premises are also valid for $\text{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I)$. We give the formation rule for $\text{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I)$ below as well as the equality rules on inductive types and constructors.

The module subtyping rules:

MSUB-SIG

$$\frac{\begin{array}{c} E; \text{Spec}_1; \dots; \text{Spec}_n[] \vdash \text{Spec}_{\sigma(i)} <: \text{Spec}'_i \text{ for } i = 1..m \\ \sigma : \{1 \dots m\} \rightarrow \{1 \dots n\} \text{ injective} \end{array}}{E[] \vdash \text{Sig } \text{Spec}_1; \dots; \text{Spec}_n \text{ End} <: \text{Sig } \text{Spec}'_1; \dots; \text{Spec}'_m \text{ End}}$$

MSUB-FUN

$$\frac{E[] \vdash T'_1 <: T_1 \quad E; \text{ModS}(X : T'_1)[] \vdash T_2 <: T'_2}{E[] \vdash \text{Funsig}(X : T_1) T_2 <: \text{Funsig}(X : T'_1) T'_2}$$

Specification subtyping rules:

ASSUM-ASSUM

$$\frac{E[] \vdash U_1 \leq_{\beta\delta\iota\zeta} U_2}{E[] \vdash \text{Assum}()(c : U_1) <: \text{Assum}()(c : U_2)}$$

DEF-ASSUM

$$\frac{E[] \vdash U_1 \leq_{\beta\delta\iota\zeta} U_2}{E[] \vdash \text{Def}()(c := t : U_1) <: \text{Assum}()(c : U_2)}$$

ASSUM-DEF

$$\frac{E[] \vdash U_1 \leq_{\beta\delta\iota\zeta} U_2 \quad E[] \vdash c =_{\beta\delta\iota\zeta} t_2}{E[] \vdash \text{Assum}()(c : U_1) <: \text{Def}()(c := t_2 : U_2)}$$

DEF-DEF

$$\frac{E[] \vdash U_1 \leq_{\beta\delta\iota\zeta} U_2 \quad E[] \vdash t_1 =_{\beta\delta\iota\zeta} t_2}{E[] \vdash \text{Def}()(c := t_1 : U_1) <: \text{Def}()(c := t_2 : U_2)}$$

IND-IND

$$\frac{E[] \vdash \Gamma_P =_{\beta\delta\iota\zeta} \Gamma'_P \quad E[\Gamma_P] \vdash \Gamma_C =_{\beta\delta\iota\zeta} \Gamma'_C \quad E[\Gamma_P; \Gamma_C] \vdash \Gamma_I =_{\beta\delta\iota\zeta} \Gamma'_I}{E[] \vdash \text{Ind}()[\Gamma_P](\Gamma_C := \Gamma_I) <: \text{Ind}()[\Gamma'_P](\Gamma'_C := \Gamma'_I)}$$

INDP-IND

$$\frac{E[] \vdash \Gamma_P =_{\beta\delta\iota\zeta} \Gamma'_P \quad E[\Gamma_P] \vdash \Gamma_C =_{\beta\delta\iota\zeta} \Gamma'_C \quad E[\Gamma_P; \Gamma_C] \vdash \Gamma_I =_{\beta\delta\iota\zeta} \Gamma'_I}{E[] \vdash \text{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I) <: \text{Ind}()[\Gamma'_P](\Gamma'_C := \Gamma'_I)}$$

INDP-INDP

$$\frac{E[] \vdash \Gamma_P =_{\beta\delta\iota\zeta} \Gamma'_P \quad E[\Gamma_P] \vdash \Gamma_C =_{\beta\delta\iota\zeta} \Gamma'_C \quad E[\Gamma_P; \Gamma_C] \vdash \Gamma_I =_{\beta\delta\iota\zeta} \Gamma'_I \quad E[] \vdash p =_{\beta\delta\iota\zeta} p'}{E[] \vdash \text{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I) <: \text{Ind}_{p'}()[\Gamma'_P](\Gamma'_C := \Gamma'_I)}$$

MODS-MODS

$$\frac{E[] \vdash T_1 <: T_2}{E[] \vdash \text{ModS}(m : T_1) <: \text{ModS}(m : T_2)}$$

MODEQ-MODS

$$\frac{E[] \vdash T_1 <: T_2}{E[] \vdash \text{ModSEq}(m : T_1 == p) <: \text{ModS}(m : T_2)}$$

MODS-MODEQ

$$\frac{E[] \vdash T_1 <: T_2 \quad E[] \vdash m =_{\beta\delta\iota\zeta} p_2}{E[] \vdash \text{ModS}(m : T_1) <: \text{ModSEq}(m : T_2 == p_2)}$$

MODEQ-MODEQ

$$\frac{E[] \vdash T_1 <: T_2 \quad E[] \vdash p_1 =_{\beta\delta\iota\zeta} p_2}{E[] \vdash \text{ModSEq}(m : T_1 == p_1) <: \text{ModSEq}(m : T_2 == p_2)}$$

MODTYPE-MODTYPE

$$\frac{E[] \vdash T_1 <: T_2 \quad E[] \vdash T_2 <: T_1}{E[] \vdash \text{ModType}(S := T_1) <: \text{ModType}(S := T_2)}$$

Verification of the specification

IMPL-SPEC

$$\frac{\mathcal{WF}(E; \text{Spec})[] \quad \text{Spec is one of Def, Assum, Ind, ModType}}{E[] \vdash \text{Spec} : \text{Spec}}$$

MOD-MODS

$$\frac{\mathcal{WF}(E; \text{ModS}(m : T))[] \quad E[] \vdash M : T}{E[] \vdash \text{Mod}(m : T := M) : \text{ModS}(m : T)}$$

MOD-MODEQ

$$\frac{\mathcal{WF}(E; \text{ModSEq}(m : T == p))[] \quad E[] \vdash p =_{\beta\delta\iota\zeta} p'}{E[] \vdash \text{Mod}(m : T := p') : \text{ModSEq}(m : T == p')}$$

New environment formation rules

WF-MODS

$$\frac{\mathcal{WF}(E)[] \quad E[] \vdash \mathcal{WF}(T)}{\mathcal{WF}(E; \text{ModS}(m : T))[]}$$

WF-MODEQ

$$\frac{\mathcal{WF}(E)[] \quad E[] \vdash p : T}{\mathcal{WF}(E, \text{ModSEq}(m : T == p))[]}$$

WF-MODTYPE

$$\frac{\mathcal{WF}(E)[] \quad E[] \vdash \mathcal{WF}(T)}{\mathcal{WF}(E, \text{ModType}(S := T))[]}$$

WF-IND

$$\frac{\begin{array}{c} \mathcal{WF}(E; \text{Ind}()[\Gamma_P](\Gamma_C := \Gamma_I))[] \\ E[] \vdash p : \text{Sig } \text{Spec}_1; \dots; \text{Spec}_n; \text{Ind}()[\Gamma'_P](\Gamma'_C := \Gamma'_I); \dots \text{ End} : \\ E[] \vdash \text{Ind}()[\Gamma'_P](\Gamma'_C := \Gamma'_I) \{p.l/l\}_{l \in \text{labels}(\text{Spec}_1; \dots; \text{Spec}_n)} <: \text{Ind}()[\Gamma_P](\Gamma_C := \Gamma_I) \end{array}}{\mathcal{WF}(E; \text{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I))[]}$$

Component access rules

ACC-TYPE

$$\frac{E[\Gamma] \vdash p : \text{Sig } \text{Spec}_1; \dots; \text{Spec}_i; \text{Assum}()(c : U); \dots \text{ End}}{E[\Gamma] \vdash p.c : U \{p.l/l\}_{l \in \text{labels}(\text{Spec}_1; \dots; \text{Spec}_i)}}$$

$$\frac{E[\Gamma] \vdash p : \text{Sig } \text{Spec}_1; \dots; \text{Spec}_i; \text{Def}()(c := t : U); \dots \text{ End}}{E[\Gamma] \vdash p.c : U \{p.l/l\}_{l \in \text{labels}(\text{Spec}_1; \dots; \text{Spec}_i)}}$$

ACC-DELTA Notice that the following rule extends the delta rule defined in section 4.3

$$\frac{E[\Gamma] \vdash p : \text{Sig } \text{Spec}_1; \dots; \text{Spec}_i; \text{Def}()(c := t : U); \dots \text{ End}}{E[\Gamma] \vdash p.c \triangleright_\delta t \{p.l/l\}_{l \in \text{labels}(\text{Spec}_1; \dots; \text{Spec}_i)}}$$

In the rules below we assume Γ_P is $[p_1 : P_1; \dots; p_r : P_r]$, Γ_I is $[I_1 : A_1; \dots; I_k : A_k]$, and Γ_C is $[c_1 : C_1; \dots; c_n : C_n]$

ACC-IND

$$\frac{\begin{array}{c} E[\Gamma] \vdash p : \text{Sig } \text{Spec}_1; \dots; \text{Spec}_i; \text{Ind}()[\Gamma_P](\Gamma_C := \Gamma_I); \dots \text{ End} \\ E[\Gamma] \vdash p.I_j : (p_1 : P_1) \dots (p_r : P_r) A_j \{p.l/l\}_{l \in \text{labels}(\text{Spec}_1; \dots; \text{Spec}_i)} \end{array}}{\begin{array}{c} E[\Gamma] \vdash p : \text{Sig } \text{Spec}_1; \dots; \text{Spec}_i; \text{Ind}()[\Gamma_P](\Gamma_C := \Gamma_I); \dots \text{ End} \\ E[\Gamma] \vdash p.c_m : (p_1 : P_1) \dots (p_r : P_r) C_m \{I_j / (I_j p_1 \dots p_r)\}_{j=1 \dots k} \{p.l/l\}_{l \in \text{labels}(\text{Spec}_1; \dots; \text{Spec}_i)} \end{array}}$$

ACC-INDP

$$\frac{E[] \vdash p : \text{Sig } \text{Spec}_1; \dots; \text{Spec}_n; \text{Ind}_{p'}()[\Gamma_P](\Gamma_C := \Gamma_I) ; \dots \text{ End}}{E[] \vdash p.I_i \triangleright_\delta p'.I_i}$$

$$\frac{E[] \vdash p : \text{Sig } \text{Spec}_1; \dots; \text{Spec}_n; \text{Ind}_{p'}()[\Gamma_P](\Gamma_C := \Gamma_I) ; \dots \text{ End}}{E[] \vdash p.c_i \triangleright_\delta p'.c_i}$$

ACC-MOD

$$\frac{E[\Gamma] \vdash p : \text{Sig } \text{Spec}_1; \dots; \text{Spec}_i; \text{ModS}(m : T); \dots \text{ End}}{E[\Gamma] \vdash p.m : T\{p.l/l\}_{l \in \text{labels}(\text{Spec}_1; \dots; \text{Spec}_i)}}$$

$$\frac{E[\Gamma] \vdash p : \text{Sig } \text{Spec}_1; \dots; \text{Spec}_i; \text{ModSEq}(m : T == p'); \dots \text{ End}}{E[\Gamma] \vdash p.m : T\{p.l/l\}_{l \in \text{labels}(\text{Spec}_1; \dots; \text{Spec}_i)}}$$

ACC-MODEQ

$$\frac{E[\Gamma] \vdash p : \text{Sig } \text{Spec}_1; \dots; \text{Spec}_i; \text{ModSEq}(m : T == p'); \dots \text{ End}}{E[\Gamma] \vdash p.m \triangleright_\delta p'\{p.l/l\}_{l \in \text{labels}(\text{Spec}_1; \dots; \text{Spec}_i)}}$$

ACC-MODTYPE

$$\frac{E[\Gamma] \vdash p : \text{Sig } \text{Spec}_1; \dots; \text{Spec}_i; \text{ModType}(S := T); \dots \text{ End}}{E[\Gamma] \vdash p.S \triangleright_\delta T\{p.l/l\}_{l \in \text{labels}(\text{Spec}_1; \dots; \text{Spec}_i)}}$$

The function $\text{labels}()$ is used to calculate the set of label of the set of specifications. It is defined by $\text{labels}(\text{Spec}_1; \dots; \text{Spec}_n) = \text{labels}(\text{Spec}_1) \cup \dots \cup \text{labels}(\text{Spec}_n)$ where $\text{labels}(\text{Spec})$ is defined as follows:

- $\text{labels}(\text{Assum}(\Gamma)(c : U)) = \{c\}$,
- $\text{labels}(\text{Def}(\Gamma)(c := t : U)) = \{c\}$,
- $\text{labels}(\text{Ind}(\Gamma)[\Gamma_P](\Gamma_C := \Gamma_I)) = \text{dom}(\Gamma_C) \cup \text{dom}(\Gamma_I)$,
- $\text{labels}(\text{ModS}(m : T)) = \{m\}$,
- $\text{labels}(\text{ModSEq}(m : T == M)) = \{m\}$,
- $\text{labels}(\text{ModType}(S := T)) = \{S\}$

Environment access for modules and module types

ENV-MOD

$$\frac{\mathcal{WF}(E; \text{ModS}(m : T); E')[\Gamma]}{E; \text{ModS}(m : T); E'[\Gamma] \vdash m : T}$$

$$\frac{\mathcal{WF}(E; \text{ModSEq}(m : T == p); E')[\Gamma]}{E; \text{ModSEq}(m : T == p); E'[\Gamma] \vdash m : T}$$

ENV-MODEQ

$$\frac{\mathcal{WF}(E; \text{ModSEq}(m : T == p); E')[\Gamma]}{E; \text{ModSEq}(m : T == p); E'[\Gamma] \vdash m \triangleright_\delta p}$$

ENV-MODTYPE

$$\frac{\mathcal{WF}(E; \text{ModType}(S := T); E')[\Gamma]}{E; \text{ModType}(S := T); E'[\Gamma] \vdash S \triangleright_\delta T}$$

ENV-INDP

$$\frac{\mathcal{WF}(E; \text{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I)) \square}{E; \text{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I) \square \vdash I_i \triangleright_\delta p.I_i}$$

$$\frac{\mathcal{WF}(E; \text{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I)) \square}{E; \text{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I) \square \vdash c_i \triangleright_\delta p.c_i}$$

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.

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 18.

6.2.4 `Opaque` *qualid*₁ ... *qualid*_n.

This 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` can only apply on constants originally defined as `Transparent`.

Constants defined by a proof ended by `Qed` are automatically stamped as `Opaque` and can no longer be considered as `Transparent`. This is to keep with the usual mathematical practice of *proof irrelevance*: what matters in a mathematical development is the sequence of lemma statements, not their actual proofs. This distinguishes lemmas from the usual defined constants, whose actual values are of course relevant in general.

See also: sections 8.5, 8.11, 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.

6.2.5 `Transparent` *qualid*₁ ... *qualid*_n.

This command is the converse of `Opaque` and can only apply on constants originally defined as `Transparent` to restore their initial behaviour after an `Opaque` command.

The constants automatically declared transparent are the ones defined by a proof ended by `Defined`, or by a `Definition` or `Local` with an explicit body.

Warning: `Transparent` and `Opaque` are not synchronous with the reset mechanism. If a constant was transparent at point A, if you set it opaque at point B and reset to point A, you return to state of point A with the difference that the constant is still opaque. This can cause changes in tactic scripts behaviour.

At section or module closing, a constant recovers the status it got at the time of its definition.

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, 8.11, 7.1.4

6.2.6 Search *qualid*.

This command displays the name and type of all theorems of the current context whose statement's conclusion has the form $(qualid\ t_1 \dots t_n)$. 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).

6.2.7 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 [*qualid-or-string* ... *qualid-or-string*].
where *qualid-or-string* is a *qualid* or a *string*.

This extension of SearchAbout searches for all objects whose statement mentions all of *qualid* of the list and whose name contains all *string* of the list.

Example:

```
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
OmegaLemmas.fast_Zmult_plus_distr:
  forall (n m p : Z) (P : Z -> Prop),
    P (n * p + m * p)%Z -> P ((n + m) * p)%Z

```

2. SearchAbout *term* inside *module*₁ ... *module*_{*n*}.
 SearchAbout [*qualid-or-string* ... *qualid-or-string*] inside *module*₁ ... *module*_{*n*}.

This restricts the search to constructions defined in modules *module*₁ ... *module*_{*n*}.

3. SearchAbout *term* outside *module*₁...*module*_{*n*}.
 SearchAbout [*qualid-or-string* ... *qualid-or-string*] outside *module*₁...*module*_{*n*}.

This restricts the search to constructions not defined in modules *module*₁ ... *module*_{*n*}.

6.2.8 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 “_”.

```

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 ... *module*_{*n*}.

2. SearchPattern *term* outside *module*₁ ... *module*_{*n*}.

This restricts the search to constructions not defined in modules *module*₁ ... *module*_{*n*}.

6.2.9 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 ... *module*_{*n*}.

2. SearchRewrite *term* outside *module*₁ ... *module*_{*n*}.

This restricts the search to constructions not defined in modules *module*₁ ... *module*_{*n*}.

6.2.10 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 (visible as O)

Coq < Locate Init.Datatypes.O.
Constructor Coq.Init.Datatypes.O (visible as O)

Coq < Locate Coq.Init.Datatypes.O.
Constructor Coq.Init.Datatypes.O (visible as O)

Coq < Locate I.Dont.Exist.
No object of suffix I.Dont.Exist
```

See also: Section 11.1.10

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

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

6.4.1 Require *dirpath* .

This command looks in the loadpath for a file containing module *dirpath*, then loads and opens (imports) its contents. More precisely, if *dirpath* splits into a library *dirpath* *dirpath*' and a module name *ident*, then the file *ident* .vo is searched in a physical path mapped to the logical path *dirpath*'.

TODO: effect on the name table.

If the module required has already been loaded, COQ simply opens it (as `Import dirpath` would do it).

If a module *A* contains a command `Require B` then the command `Require A` loads the module *B* but does not open it (See the `Require Export` variant below).

Variants:

1. Require Export *qualid* .
This command acts as `Require qualid`. But if a module *A* contains a command `Require Export B`, then the command `Require A` opens the module *B* as if the user would have typed `Require B`.
2. Require *qualid* *string* .
Specifies the file to load as being *string* but containing module *qualid* which is then opened.

These different variants can be combined.

Error messages:

1. Cannot load *ident*: no physical path bound to *dirpath*

2. Can't find module toto on loadpath

The command did not find the file `toto.vo`. Either `toto.v` exists but is not compiled or `toto.vo` is in a directory which is not in your `LoadPath` (see section 6.5).

3. 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.

See also: chapter 12

6.4.2 Print Modules.

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

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 12).

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 page 117)

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 2.5.2).

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 path *string* to the current COQ loadpath and maps it to the logical directory *dirpath*, which means that every file *M.v* physically lying in directory *string* becomes accessible through logical name “*dirpath.M*”.

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 directory *string* and all its subdirectories to the current COQ loadpath. The top directory *string* is mapped to the logical directory *dirpath* while any subdirectory *pdir* is mapped to logical directory *dirpath.pdir* and so on.

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.

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 `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.

6.6.4 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 115).

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`, `judgements`, 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 page 213).
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 12.4).

6.7.3 Time *command*.

This command executes the vernac 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.

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* [68, 6, 64, 71], 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. Local *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).

Variants:

1. 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` .

2. 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 is ended by “...”. In this case the tactic command typed by the user is equivalent to `command;tactic`.

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 cancelling the current proofs.

7.1.8 Resume .

This command 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

7.2 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 Displaying 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`.

7.3.2 `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.

7.3.3 `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`

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 CIC 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`

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 ident1 ... identn.`
This is equivalent to `clear ident1. ... clear identn.`
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.

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.

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 context¹

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 4.2². 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 $T \rightarrow U$, then it puts in the local context either $\text{H}n:T$ (if T is of type `Set` or `Prop`) or $\text{X}n:T$ (if the type of T is `Type`). The optional index n is such that $\text{H}n$ or $\text{X}n$ 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.

¹but it does not rename the hypothesis in the proof-term...

²Actually, only the second subgoal will be generated since the other one can be automatically checked.

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 2.5.2).

3. `intros ident1 ... identn`

Is equivalent to the composed tactic `intro ident1; ... ; intro identn`.

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 premise. For instance, on the subgoal `forall x y:nat, x=y -> forall z:nat, z=x->z=y` the tactic `intros until 2` is equivalent to `intros x y H z H0` (assuming *x*, *y*, *H*, *z* and *H0* do not already occur in context).

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`

Applies `intro` but puts the introduced hypothesis after the hypothesis *ident* in the hypotheses.

Error messages:

- (a) No product even after head-reduction
- (b) No such hypothesis: *ident*

7. `intro ident1 after ident2`

Behaves as previously but *ident₁* is the name of the introduced hypothesis. It is equivalent to `intro ident1; move ident1 after ident2`.

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*. The tactic `apply` relies on first-order pattern-matching with dependent types. See `pattern` in section 8.5.7 to transform a second-order pattern-matching problem into a first-order one.

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, 8.3.10).

2. generated subgoal *term'* has metavariables in it

This occurs when some instantiations of 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 term1 ... termn`

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 *term₁ ... 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 (ref1 := term1) ... (refn := termn)`

This also provides `apply` with values for instantiating premises. But variables are referred by names and non dependent products by order (see syntax in Section 8.3.11).

3. `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.

4. `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* \rightarrow *B* with *B* possibly containing products. Then it generates two subgoals *B* \rightarrow *G* and *A*. Applying `lapply H` (where *H* has type *A* \rightarrow *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 *`
`set (ident := term) in * | - *`

This behaves as above but substitutes *term* everywhere in the goal (both in conclusion and hypotheses).

2. `set (ident := term) in * | -`

This behaves the same but substitutes *term* in the hypotheses only (not in the conclusion).

3. `set (ident := term) in | - *`

This is equivalent to `set (ident := term)`, i.e. it substitutes *term* in the conclusion only.

4. `set (ident0 := term) in ident1`

This behaves the same but substitutes *term* only in the hypothesis named *ident₁*.

5. `set (ident0 := term) in ident1 at num1 ... numn`

This notation allows to specify which occurrences of *term* have to be substituted in the hypothesis named *ident₁*. The occurrences are numbered from left to right and are meaningful on a pure expression using no implicit argument, notation or coercion. A negative occurrence number means an occurrence which should not be substituted. As an exception of the left-to-right order, the occurrences in the `return` subexpression of a `match` are considered *before* the occurrences in the matched term.

For expressions using notations, or hiding implicit arguments or coercions, it is recommended to make explicit all occurrences in order by using `Set Printing All` (see section 2.8).

6. `set (ident := term) in | - * at num1 ... numn`

This allows to specify which occurrences of the conclusion are concerned.

7. `set (ident0 := term) in ident1 at num11 ... numn11, ... identm at num1m ... numnmm`

It substitutes *term* at occurrences *num₁ⁱ ... num_{n_i}ⁱ* of hypothesis *ident_i*. Each `at` part is optional.

8. `set (ident0 := term) in ident1 at num11 ... numn11, ... identm at num1m ... numnmm | - *
at num1' ... numn'`

This is the more general form which combines all the previous possibilities.

9. `set term`

This behaves as `set (ident := term)` but *ident* is generated by COQ. This variant is available for the forms with `in` too.

10. `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.

11. `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: $U \rightarrow T$ and *U*. The subgoal $U \rightarrow T$ comes first in the list of remaining subgoal to prove.

8.3.9 `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 accordingly to *T*.

Variants:

³This corresponds to the cut rule of sequent calculus.

1. `generalize term1 ... termn`

Is equivalent to `generalize termn; ... ; generalize term1`. Note that the sequence of `termi`'s are processed from n to 1.

2. `generalize dependent term`

This generalizes `term` but also *all* hypotheses which depend on `term`. It clears the generalized hypotheses.

8.3.10 `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 term1 with term2`

This replaces the occurrences of `term1` by `term2` in the current goal. The terms `term1` and `term2` must be convertible.

2. `change term1 at num1 ... numi with term2`

This replaces the occurrences numbered `num1 ... numi` of `term1` by `term2` in the current goal. The terms `term1` and `term2` must be convertible.

Error message: Too few occurrences

3. `change term in ident`

4. `change term1 with term2 in ident`

5. `change term1 at num1 ... numi with term2 in ident`

This applies the change tactic not to the goal but to the hypothesis `ident`.

See also: 8.5

8.3.11 Bindings list

A bindings list is generally used after the keyword `with` in tactics. The general shape of a bindings list is `(ref1 := term1) ... (refn := termn)` where `ref` is either an `ident` or a `num`. It is used to provide a tactic with a list of values `(term1, ..., termn)` that have to be substituted respectively to `ref1, ..., refn`. For all $i \in [1 \dots n]$, if `refi` is `identi` then it references the dependent product `identi : T` (for some type `T`); if `refi` is `numi` then it references the `numi`-th non dependent premise.

A bindings list can also be a simple list of terms `term1 term2 ... termn`. In that case the references to which these terms correspond are determined by the tactic. In case of `elim` (see section 4) the terms should correspond to all the dependent products in the type of `term` while in the case of `apply` only the dependent products which are not bound in the conclusion of the type are given.

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 $\sim P$ and P . It is very useful in proofs by cases, where some cases are impossible. In most cases, P or $\sim P$ 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 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

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. The specification of such parts are called *clauses*. It can be either the conclusion, or an hypothesis. In the case of a defined hypothesis it is possible to specify if the conversion should occur on the type part, the body part or both (default).

Clauses are written after a conversion tactic (tactic `set` 8.3.7 also uses clauses) and are introduced by the keyword `in`. If no clause is provided, the default is to perform the conversion only in the conclusion.

The syntax and description of the various clauses follows:

in $H_1 \dots H_n$ **| -** only in hypotheses $H_1 \dots H_n$

in $H_1 \dots H_n$ **| - *** in hypotheses $H_1 \dots H_n$ and in the conclusion

in ***** **| -** in every hypothesis

in ***** (equivalent to **in** ***** **| - ***) everywhere

in (**type of** H_1) (**value of** H_2) \dots **| -** in type part of H_1 , in the value part of H_2 , etc.

For backward compatibility, the notation `in $H_1 \dots H_n$` performs the conversion in hypotheses $H_1 \dots H_n$.

8.5.1 cbv $flag_1 \dots flag_n$, lazy $flag_1 \dots flag_n$ and compute

These parameterized reduction tactics apply to any goal and perform the normalization of the goal according to the specified flags. Since the reduction considered in COQ include β (reduction of functional application), δ (unfolding of transparent constants, see 6.2.5), ι (reduction of `Cases`, `Fix` and `CoFix` expressions) and ζ (removal of local definitions), every flag is one of `beta`, `delta`, `iota`, `zeta`, `[qualid1 . . . qualidk]` and `-[qualid1 . . . qualidk]`. The last two flags give the list of constants to unfold, or the list of constants not to unfold. These two flags can occur only after the `delta` flag. If alone (i.e. not followed by `[qualid1 . . . qualidk]` or `-[qualid1 . . . qualidk]`), the `delta` flag means that all constants must be unfolded. However, the `delta` flag does not apply to variables bound by a `let-in`

construction whose unfolding is controlled by the `zeta` flag only. In addition, there is a flag `Evar` to perform instantiation of existential variables (“?”) when an instantiation actually exists.

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, but if an argument is used several times, it is computed only once. This reduction is efficient for reducing expressions with dead code. For instance, the proofs of a proposition $\exists_T x.P(x)$ reduce to a pair of a witness t , and a proof that t verifies 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 should be preferred for evaluating purely computational expressions (i.e. with few dead code).

Variants:

1. `compute`

This tactic is an alias for `cbv beta delta evar iota zeta`.

Error messages:

1. Delta must be specified before

A list of constants appeared before the `delta` flag.

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. `hnf` does not produce a real head normal form but either a product or an applicative term in head normal form or a variable.

Example: The term `forall n:nat, (plus (S n) (S n))` is not reduced by `hnf`.

Remark: The δ rule only applies to transparent constants (see section 6.2.4 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 0)=n` does change nothing.

Variants:

1. `simpl term`

This applies `simpl` only to the occurrences of *term* in the current goal.

2. `simpl term at num1 ... numi`

This applies `simpl` only to the num_1, \dots, 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 num1 ... numi`

This applies `simpl` only to the num_1, \dots, 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 6.2.5). 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 qualid1, ..., qualidn`

Replaces *simultaneously* $qualid_1, \dots, qualid_n$ with their definitions and replaces the current goal with its $\beta\iota$ normal form.

2. `unfold qualid1 at num11, ..., numi1, ..., qualidn at num1n ... numjn`

The lists num_1^1, \dots, num_i^1 and num_1^n, \dots, num_j^n specify the occurrences of $qualid_1, \dots, qualid_n$ 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

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 substituted for *term*.

Variants:

1. `fold term1 ... termn`

Equivalent to `fold term1; ... ; fold termn`.

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 as $\phi(t)$ where the notation captures all the instances of t in $\phi(t)$, then `pattern t` transforms it into $(\text{fun } x:A \Rightarrow \phi(x))\ t$. This command can be used, for instance, when the tactic `apply` fails on matching.

Variants:

1. `pattern term at num1 ... numn`

Only the occurrences $\text{num}_1 \dots \text{num}_n$ of *term* will be considered for β -expansion. Occurrences are located from left to right.

2. `pattern term1, ..., termm`

Starting from a goal $\phi(t_1 \dots t_m)$, the tactic `pattern t1, ..., tm` generates the equivalent goal $(\text{fun } (x_1:A_1) \dots (x_m:A_m) \Rightarrow \phi(x_1 \dots x_m))\ t_1 \dots t_m$.

If t_i occurs in one of the generated types A_j these occurrences will also be considered and possibly abstracted.

3. `pattern term1 at num11 ... numn11, ..., termm at num1m ... numnmm`

This behaves as above but processing only the occurrences $\text{num}_1^1, \dots, \text{num}_i^1$ of $\text{term}_1, \dots, \text{num}_1^m, \dots, \text{num}_j^m$ of term_m starting from term_m .

8.5.8 Conversion tactics applied to hypotheses

`conv_tactic in ident1 ... identn`

Applies the conversion tactic *conv_tactic* to the hypotheses $\text{ident}_1, \dots, \text{ident}_n$. 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 c_i 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 is the number of constructors of the head of the goal.

2. `constructor num with bindings_list`

Let c_i 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 $\exists x.P(x)$. 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`.

8.7 Eliminations (Induction and Case Analysis)

Elimination tactics are useful to prove statements by induction or case analysis. Indeed, they make use of the elimination (or induction) principles generated with inductive definitions (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. Cannot refine to conclusions with meta-variables

As `induction` uses `apply`, see Section 8.3.6 and the variant `elim ... with ...` below.

Variants:

1. `induction term as intro_pattern`

This behaves as `induction term` but uses the names in `intro_pattern` to name the variables introduced in the context. The `intro_pattern` must have the form $[p_{11} \dots p_{1n_1} \mid \dots \mid p_{m1} \dots p_{mn_m}]$ 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 $p_{i1} \dots p_{in_i}$ in order. If there are not enough names, `induction` invents names for the remaining variables to introduce. More generally, the p 's can be any introduction patterns (see Section 8.7.3). This provides a concise notation for nested induction.

Remark: for an inductive type with one constructor, the pattern notation (p_1, \dots, p_n) can be used instead of $[p_1 \dots p_n]$.

2. `induction term using qualid`

This behaves as `induction term` but using the induction scheme of name `qualid`. It does not expect that the type of `term` is inductive.

3. `induction term using qualid as intro_pattern`

This combines `induction term using qualid` and `induction term as intro_pattern`.

4. `elim term`

This is a more basic induction tactic. Again, the type of the argument `term` must be an inductive constant. Then according to the type of the goal, the tactic `elim` chooses the right destructor and applies it (as in the case of the `apply` tactic). For instance, assume that our proof context contains `n:nat`, assume that our current goal is `T` of type `Prop`, then `elim n` is equivalent to `apply nat_ind with (n:=n)`. The tactic `elim` does not affect the hypotheses of the goal, neither introduces the induction loading into the context of hypotheses.

5. `elim term`

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

6. `elim term with term1 ... termn`

Allows the user to give explicitly the values for dependent premises of the elimination schema. All arguments must be given.

Error message: Not the right number of dependent arguments

7. `elim term with ref1 := term1 ... refn := termn`

Provides also `elim` with values for instantiating premises by associating explicitly variables (or non dependent products) with their intended instance.

8. `elim term1 using term2`

Allows the user to give explicitly an elimination predicate `term2` which is not the standard one for the underlying inductive type of `term1`. Each of the `term1` and `term2` is either a simple term or a term with a bindings list (see 8.3.11).

9. `elimtype form`

The argument *form* must be inductively defined. `elimtype I` is equivalent to `cut I. intro Hn; elim Hn; clear Hn`. Therefore the hypothesis *Hn* 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`.

Error message: Impossible to unify ... with ...

Arises when *form* needs to be applied to parameters.

10. `simple induction ident`

This tactic behaves as `intros until ident; elim ident` when *ident* is a quantified variable of the goal.

11. `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 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 `[p11 ... p1n1 | ... | pm1 ... pmnm]` 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} ... p_{in_i}* in order. If there are not enough names, `destruct` invents names for the remaining variables to introduce. More generally, the *p*'s can be any introduction patterns (see Section 8.7.3). This provides a concise notation for nested destruction.

Remark: for an inductive type with one constructor, the pattern notation `(p1, ..., pn)` can be used instead of `[p1...pn]`.

2. `destruct term using qualid`

This is a synonym of `induction term using qualid`.

3. `destruct term as intro_pattern using qualid`

This is a synonym of `induction term using qualid as intro_pattern`.

4. `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.

5. `case term with term1 ... termn`

Analogous to `elim ... with above`.

6. `simple destruct ident`

This tactic behaves as `intros until ident; case ident` when `ident` is a quantified variable of the goal.

7. `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`

The tactic `intros` applied to introduction patterns performs both introduction of variables and case analysis in order to give names to components of an hypothesis.

An introduction pattern is either:

- the wildcard: `_`
- a variable
- a disjunction of lists of patterns: `[p11 ... p1m1 | ... | p11 ... pnmn]`
- a conjunction of patterns: `(p1 , ... , pn)`

The behavior of `intros` is defined inductively over the structure of the pattern given as argument:

- introduction on the wildcard do the introduction and then immediately clear (cf 8.3.2) the corresponding hypothesis;
- introduction on a variable behaves like described in 8.3.5;
- introduction over a list of patterns `p1 ... pn` is equivalent to the sequence of introductions over the patterns namely: `intros p1; ...; intros pn`, the goal should start with at least `n` products;
- introduction over a disjunction of list of patterns `[p11 ... p1m1 | ... | p11 ... pnmn]`. It introduces a new variable `X`, its type should be an inductive definition with `n` constructors, then it performs a case analysis over `X` (which generates `n` subgoals), it clears `X` and performs on each generated subgoals the corresponding `intros pi1 ... pimi` tactic;
- introduction over a conjunction of patterns `(p1, ..., pn)`, it introduces a new variable `X`, its type should be an inductive definition with 1 constructor with (at least) `n` arguments, then it performs a case analysis over `X` (which generates 1 subgoal with at least `n` products), it clears `X` and performs an introduction over the list of patterns `p1 ... pn`.

Remark: The pattern `(p1, ..., pn)` is a synonym for the pattern `[p1 ... pn]`, i.e. it corresponds to the decomposition of an hypothesis typed by an inductive type with a single constructor.

```

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 < Proof 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 $\text{forall } n\ m:\text{nat}, P\ n\ m$ then, `double induction n m` yields the four cases with their respective inductive hypotheses. In particular the case for $(P\ (S\ n)\ (S\ m))$ with the induction hypotheses $(P\ (S\ n)\ m)$ and $(m:\text{nat})(P\ n\ m)$ (hence $(P\ n\ m)$ and $(P\ n\ (S\ m))$).

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 an *num* is valid.

8.7.5 decompose [$qualid_1 \dots qualid_n$] *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 p. 43).

8.7.6 functional induction *ident term₁ ... term_n*.

The *experimental* tactic `functional induction` performs case analysis and induction following the definition of a (not mutually recursive) function.

```

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.

```

`functional induction` is a shorthand for the more general command `Functional Scheme` which builds induction principles following the recursive structure of (possibly mutually recursive) functions. **See also:** 10.4 for the difference between using one or the other.

Remark: `functional induction` may fail on functions built by tactics. In particular case analysis of a function are considered only if they are not inside an application.

Remark: Arguments of the function must be given, including the implicit ones. If the function is recursive, arguments must be variables, otherwise they may be any term.

See also: 8.14,10.4

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

$$(x_1:A_1) \dots (x_n:A_n) \text{term}_1 = \text{term}_2.$$

Then `rewrite term` replaces every occurrence of term_1 by term_2 in the goal. Some of the variables x_1 are solved by unification, and some of the types A_1, \dots, A_n become new subgoals.

Remark: In case the type of term_1 contains occurrences of variables bound in the type of *term*, the tactic tries first to find a subterm of the goal which matches this term in order to find a closed instance term'_1 of term_1 , and then all instances of term'_1 will be replaced.

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 $\text{term}_1 = \text{term}_2$ from right to left
3. `rewrite term in ident`
Analogous to `rewrite term` but rewriting is done in the hypothesis named *ident*.
4. `rewrite -> term in ident`
Behaves as `rewrite term in ident`.
5. `rewrite <- term in ident`
Uses the equality $\text{term}_1 = \text{term}_2$ from right to left to rewrite in the hypothesis named *ident*.

8.8.2 `cutrewrite -> term1 = term2`

This tactic acts like `replace term1 with term2` (see below).

8.8.3 `replace term1 with term2`

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 $\text{term}_2 = \text{term}_1$ 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 term2=term1; [intro Hn; rewrite <- Hn; clear Hn| assumption || symmetry; try assumption]`.

Variants:

1. `replace term1 with term2 in ident`
This replaces term_1 with term_2 in the hypothesis named *ident*, and generates the subgoal $\text{term}_2 = \text{term}_1$.

Error messages:

- (a) No such hypothesis: *ident*
- (b) Nothing to rewrite in *ident*

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 ident1 ... identn`
Is equivalent to `subst ident1; ...; subst identn`.
2. `subst`
Applies `subst` repeatedly to all identifiers from the context for which an equality exists.

8.8.8 `step1 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 `step1 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 qualid.
```

where *qualid* is the name of the lemma.

The tactic is especially useful for parametric setoids which are not accepted as regular setoids for `rewrite` and `setoid_replace` (see chapter 20).

Variants:

1. `step1 term by tactic`
This applies `step1 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 qualid.`

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 term1 term2 .`

Solves a goal of the form `{term1=term2} + {~term1=term2}`.

8.9.2 `compare term1 term2`

This tactic compares two given objects `term1` and `term2` of an inductive datatype. If `G` is the current goal, it leaves the sub-goals `term1=term2 -> G` and `~term1=term2 -> G`. The type of `term1` and `term2` must satisfy the same restrictions as in the tactic `decide equality`.

8.9.3 `discriminate ident`

This tactic proves any goal from an absurd hypothesis stating that two structurally different terms of an inductive set are equal. For example, from the hypothesis `(S (S O)) = (S O)` we can derive by absurdity any proposition. Let `ident` be a hypothesis of type `term1 = term2` in the local context, `term1` and `term2` being elements of an inductive set. To build the proof, the tactic traverses the normal forms⁴ of `term1` and `term2` looking for a couple of subterms `u` and `w` (`u` subterm of the normal form of `term1` and `w` subterm of the normal form of `term2`), 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: If `ident` does not denote an hypothesis in the local context but refers to an hypothesis quantified in the goal, then the latter is first introduced in the local context using `intros until ident`.

Error messages:

1. `ident Not a discriminable equality`

occurs when the type of the specified hypothesis is not an equation.

Variants:

⁴Recall: opaque constants will not be expanded by δ reductions

1. `discriminate num`

This does the same thing as `intros until num` then `discriminate ident` where `ident` is the identifier for the last introduced hypothesis.

2. `discriminate`

It applies to a goal of the form $\sim term_1 = term_2$ and it is equivalent to: `unfold not; intro ident; discriminate ident`.

Error messages:

- (a) No discriminable equalities
occurs when the goal does not verify the expected preconditions.

8.9.4 `injection ident`

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 `ident` is an hypothesis of type $term_1 = term_2$, then `injection` behaves as applying injection as deep as possible to derive the equality of all the subterms of $term_1$ and $term_2$ placed in the same positions. For example, from the hypothesis $(S (S n)) = (S (S (S m)))$ we may derive $n = (S m)$. To use this tactic $term_1$ and $term_2$ should be elements of an inductive set and they should be neither explicitly equal, nor structurally different. We mean by this that, if 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 couple of subterms u and w , u subterm of n_1 and w subterm of n_2 , placed in the same positions and having different constructors as head symbols.

If these conditions are satisfied, then, the tactic derives the equality of all the subterms of $term_1$ and $term_2$ placed in the same positions and puts them as antecedents of the current goal.

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: If *ident* does not denote an hypothesis in the local context but refers to an hypothesis quantified in the goal, then the latter is first introduced in the local context using `intros until ident`.

Error messages:

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

Variants:

1. `injection num`

This does the same thing as `intros until num` then `injection ident` where *ident* is the identifier for the last introduced hypothesis.

2. `injection`

If the current goal is of the form $term_1 <> term_2$, the tactic computes the head normal form of the goal and then behaves as the sequence: `unfold not; intro ident; injection ident`.

Error message: goal does not satisfy the expected preconditions

8.9.5 `simplify_eq ident`

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

Remark: If *ident* does not denote an hypothesis in the local context but refers to an hypothesis quantified in the goal, then the latter is first introduced 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` If the current goal has form $\sim t_1 = t_2$, then this tactic does `hnf; intro ident; simplify_eq ident`.

8.9.6 `dependent rewrite -> ident`

This tactic applies to any goal. If *ident* has type $(\text{exists } A \ B \ a \ b) = (\text{exists } A \ B \ a' \ b')$ 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 an hypothesis in the local context but refers to an 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 $[p_{11} \dots p_{1n_1} \mid \dots \mid p_{m1} \dots p_{mn_m}]$ 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 $p_{i1} \dots p_{in_i}$ in order. If there are not enough names, `inversion` 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 $[p_{ij1} \dots p_{ijq}]$ (or, equivalently, $(p_{ij1}, \dots, p_{ijq})$) where q is the number of subequations 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 (l :: 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 ident1 ... identn`

Let $ident_1 \dots ident_n$, be identifiers in the local context. This tactic behaves as generalizing $ident_1 \dots ident_n$, and then performing `inversion`.

7. `inversion ident as intro_pattern in ident1 ... identn`

This allows to name the hypotheses introduced in the context by `inversion ident in ident1 ... identn`.

8. `inversion_clear ident in ident1 ... identn`

Let $ident_1 \dots ident_n$, be identifiers in the local context. This tactic behaves as generalizing $ident_1 \dots ident_n$, and then performing `inversion_clear`.

9. `inversion_clear ident as intro_pattern in ident1 ... identn`

This allows to name the hypotheses introduced in the context by `inversion_clear ident in ident1 ... identn`.

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 allow to give the good 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` do.

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 ident1... identn`

This tactic behaves as generalizing *ident₁... 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 `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 build 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 10.7 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 [ident1 ... identn]`
All terms that are build only with `ident1 ... identn` will be considered by `quote` as constants rather than variables.

8.11 Automatizing

8.11.1 `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 ident1 ... identn`
Uses the hint databases `ident1 ... identn` in addition to the database `core`. See Section 8.12 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 8.12
4. `trivial`
This tactic is a restriction of `auto` that is not recursive and tries only hints which cost is 0. Typically it solves trivial equalities like $X = X$.
5. `trivial with ident1 ... identn`
6. `trivial with *`

Remark: `auto` either solves completely the goal or else leave it intact. `auto` and `trivial` never fail.

See also: Section 8.12

8.11.2 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 8.12

8.11.3 tauto

This tactic implements a decision procedure for intuitionistic propositional calculus based on the contraction-free sequent calculi LJ^T* of Roy Dyckhoff [49]. 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 $(\text{forall } x:\text{nat}, \sim A \rightarrow P\ x)$ cannot be treated as atomic and an instantiation of x is necessary.

8.11.4 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 [89]. 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 reengineered by David Delahaye using mainly the tactic language (see chapter 9). The code is now quite shorter and a significant increase in performances 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 `*`.

8.11.5 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 ident1 ... identn`
Adds lemmas `ident1 ... identn` 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.

8.11.6 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.

```
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 (B:=A) a -> Some (f c) = Some (f d) -> c = d
```

```
Coq < intros.
1 subgoal
```

```
H : f = pair (B:=A) a
H0 : Some (f c) = Some (f d)
=====
c = d
```

```
Coq < congruence.
Proof completed.
```

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 managed to find a proof of the goal or of a discriminable equality.

8.11.7 `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 and inequalities 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` (chapter 17).

8.11.8 `ring term1 ... termn`

This tactic, written by Samuel Boutin and Patrick Loiseleur, applies associative commutative rewriting on every ring. The tactic must be loaded by `Require Import Ring`. The ring must be declared in the `Add Ring` command (see 19). The ring of booleans is predefined; if one wants to use the tactic on `nat` one must first require the module `ArithRing`; for `Z`, do `Require Import ZArithRing`; for `N`, do `Require Import NArithRing`.

The terms `term1, ..., termn` 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. `ring` When the goal is an equality $t_1 = t_2$, it acts like `ring t1 t2` and then simplifies or solves the equality.
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 0) n`.

Example:

```
Coq < Require Import ZArithRing.
Coq < Goal forall a b c : Z,
Coq <   (a + b + c) * (a + b + c) =
Coq <   a * a + b * b + c * c + 2 * a * b + 2 * a * c + 2 * b * c.
1 subgoal

=====
forall a b c : Z,
(a + b + c) * (a + b + c) =
a * a + b * b + c * c + 2 * a * b + 2 * a * c + 2 * b * c

Coq < intros; ring.
Proof completed.
```

You can have a look at the files `Ring.v`, `ArithRing.v`, `ZArithRing.v` to see examples of the `Add Ring` command.

See also: Chapter 19 for more detailed explanations about this tactic.

8.11.9 `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 Field`. Field theories are declared (as for `ring`) with the `Add Field` command.

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 * ((x + y) * y))%R <> 0%R

```

8.11.10 Add Field

This vernacular command adds a commutative field theory to the database for the tactic `field`. You must provide this theory as follows:

```
Add 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 $A \rightarrow A \rightarrow A$, $Amult$ is a term of type $A \rightarrow A \rightarrow A$, $Aone$ is a term of type A , $Azero$ is a term of type A , $Aopp$ is a term of type $A \rightarrow A$, Aeq is a term of type $A \rightarrow \text{bool}$, $Ainv$ is a term of type $A \rightarrow A$, Rth is a term of type $(\text{Ring_Theory } A \ Aplus \ Amult \ Aone \ Azero \ Ainv \ Aeq)$, and $Tinvl$ is a term of type $\text{forall } n:A, \sim(n=Azero) \rightarrow (Amult \ (Ainv \ n) \ n) = Aone$. To build a ring theory, refer to Chapter 19 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 19). 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 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 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: file `theories/Reals/Rbase.v` for an example of instantiation,
theory `theories/Reals` for many examples of use of `field`.

See also: [37] for more details regarding the implementation of `field`.

8.11.11 fourier

This tactic written by Loïc Pottier solves linear inequations on real numbers using Fourier's method [58]. 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.

```

8.11.12 autorewrite with $ident_1 \dots ident_n$.

This tactic ⁵ carries out rewritings according the rewriting rule bases $ident_1 \dots ident_n$.

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 $ident_1 \dots ident_n$ using tactic`
Performs, in the same way, all the rewritings of the bases $ident_1 \dots ident_n$ applying *tactic* to the main subgoal after each rewriting step.

8.11.13 Hint Rewrite $term_1 \dots term_n : ident$

This vernacular command adds the terms $term_1 \dots term_n$ (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 2.3): 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 -> $term_1 \dots term_n : ident$`
This is strictly equivalent to the command above (we only make explicit the orientation which otherwise defaults to `->`).
2. `Hint Rewrite <- $term_1 \dots term_n : ident$`
Adds the rewriting rules $term_1 \dots term_n$ with a right-to-left orientation in the base *ident*.
3. `Hint Rewrite $term_1 \dots term_n$ using tactic : ident`
When the rewriting rules $term_1 \dots term_n$ in *ident* will be used, the tactic *tactic* will be applied to the generated subgoals, the main subgoal excluded.

See also: 10.6 for examples showing the use of this tactic.

⁵The 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 reengineering of the code, this tactic has also been completely revised to get a very compact and readable version.

8.12 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 8.12.2). Each hint has a cost that is a nonnegative integer, and a 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. The general command to add a hint to some database *ident*₁, ..., *ident*_{*n*} is:

`Hint hint_definition : ident1 ... identn`

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* does contain 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 8.11.2). A typical example of 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 term1 ... termm`

Adds each `Resolve termi`.

- **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 that the symmetry of equality or $n + 1 = m + 1 \rightarrow n = m$ that we may like to introduce with a limited use in order to avoid useless proof-search.

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

Error messages:

1. `Bound head variable`

2. *term* cannot be used as a hint

Variants:

1. Immediate *term*₁ ... *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*}.

- 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, a 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 succeed 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 a0 p; decide equality.
   generalize b0 n0; decide equality.

   generalize a1 n; decide equality.

Proof completed.

```

Remark: There is currently (in the 8.0 release) no way to do pattern-matching on hypotheses.

Variants:

1. Hint *hint_definition*

No database name is given : the hint is registered in the `core` database.

2. Hint `Local hint_definition : ident1 ... identn`

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.

8.12.1 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 proven 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 inequations 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 proven in the `Lists` subdirectory.

`sets` contains lemmas about sets and relations from the directories `Sets` and `Relations`.

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.

8.12.2 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, to 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*.

8.12.3 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 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 ident1 := Induction for ident'1 Sort sort1 with ...`

ident'₁ ... ident'_m are different inductive type identifiers belonging to the same package of mutual inductive definitions. This command generates *ident₁ ... 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 ident1 := Minimality for ident'1 Sort sort1
with`

`...`

`with identm := Minimality for ident'm Sort sortm`

Same as before but defines a non-dependent elimination principle more natural in case of inductively defined relations.

See also: 10.3

See also: Section 10.3

8.14 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` $ident_i$:= Induction for $ident'_i$ with $ident'_1 \dots ident'_m$.

$ident'_1 \dots ident'_m$ are the names of mutually recursive functions (they must be in the same order as when they were defined), $ident'_i$ being one of them. This command generates the induction principle $ident_i$, following the recursive structure and case analyses of the functions $ident'_1 \dots ident'_m$, and having $ident'_i$ as entry point.

Variants:

1. `Functional Scheme` $ident_1$:= Induction for $ident'_1$.

This command is a shortcut for:

`Functional Scheme` $ident_1$:= Induction for $ident'_1$ with $ident'_1$.

This variant can be used for non mutually recursive functions only.

See also: Section 10.4

8.15 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 2.3) 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.

The 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 [36]. 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 page 25 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 identifiers 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 there can be specific variables like *?id* where *id* is a *ident* or *_*, which are metavariables for pattern matching. *?id* 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, they are used without the question mark.

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. 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 tactic1 | | tactic2 ; tactic3 ; [ tactic31 | . . . | tactic3n ] ; tactic4 .
```

is understood as

```
(try (repeat (tactic1 | | tactic2))) ;  
( ( tactic3 ; [ tactic31 | . . . | tactic3n ] ) ; tactic4 ) .
```

```

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 let_clause with ... with let_clause in atom
           | let rec rec_clause with ... with rec_clause in expr
           | 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
           | abstract atom
           | abstract atom using ident
           | first [ expr | ... | expr ]
           | solve [ expr | ... | expr ]
           | idtac | idtac string
           | fail | fail natural string
           | fresh | fresh string
           | context ident [ term ]
           | eval redexpr in term
           | type of term
           | constr : term
           | atomic_tactic
           | qualid tacarg ... tacarg
           | atom

atom ::= qualid
       | ()
       | ( expr )

```

Figure 9.1: Syntax of the tactic language

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 local definition or of an application).

<i>tacarg</i>	$::=$	<i>qualid</i> () <i>ltac</i> : <i>atom</i> <i>term</i>
<i>let_clause</i>	$::=$	<i>ident</i> [<i>name</i> ... <i>name</i>] := <i>expr</i>
<i>rec_clause</i>	$::=$	<i>ident</i> <i>name</i> ... <i>name</i> := <i>expr</i>
<i>context_rule</i>	$::=$	<i>context_hyps</i> , ... , <i>context_hyps</i> - <i>cpattern</i> => <i>expr</i> - <i>cpattern</i> => <i>expr</i> _ => <i>expr</i>
<i>context_hyps</i>	$::=$	<i>name</i> : <i>cpattern</i>
<i>match_rule</i>	$::=$	<i>cpattern</i> => <i>expr</i> <i>context</i> [<i>ident</i>] [<i>cpattern</i>] => <i>expr</i> _ => <i>expr</i>

Figure 9.2: Syntax of the tactic language (continued)

<i>top</i>	$::=$	<i>Ltac</i> <i>ltac_def</i> with ... with <i>ltac_def</i>
<i>ltac_def</i>	$::=$	<i>ident</i> [<i>ident</i> ... <i>ident</i>] := <i>expr</i>

Figure 9.3: Tactic toplevel definitions

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 associating.

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 *i* = 1, ..., *n*. It fails if the application of *v*₀ does not generate exactly *n* subgoals.

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.

Repeat loop

We have a repeat loop with:

```
repeat  $expr$ 
```

$expr$ is evaluated to v . v must be a tactic value. v is applied until it fails. Supposing $n > 1$, after the first application of v , v is applied, at least once, to the generated subgoals and so on. It stops when it fails for all the generated subgoals. It 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 then v_2 is applied. Branching is left associating.

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 $i = 1, \dots, n$. 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 [ expr1 | ... | exprn ]
```

*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.

```
idtac and idtac "message"
```

The latter variant prints the string on the standard output.

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 caught by `try` or `match goal`. There are three variants:

```
fail n, fail "message" and fail n "message"
```

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.

Error message: Tactic Failure "message" (level *n*).

Local definitions

Local definitions can be done as follows:

```
let ident1 := expr1
with ident2 := expr2
...
with identn := exprn 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`. Only functions can be defined by recursion, so at least one argument is required.

Application

An application is an expression of the following form:

```
qualid tacarg1 ... tacargn
```

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 ident1 ... identn => 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
| cpattern1 => expr1
| cpattern2 => expr2
...
| cpatternn => exprn
| _ => exprn+1
end
```

The *expr* is evaluated and should yield a term which is matched (non-linear first order unification) against *cpattern*₁ then *expr*₁ is evaluated into some value by substituting the pattern matching instantiations to the metavariables. If the matching with *cpattern*₁ fails, *cpattern*₂ is used and so on. The pattern `_` matches any term and shunts all remaining patterns if any. If *expr*₁ evaluates to a tactic, this tactic is not immediately applied to the current goal (in contrast with `match goal`). If all clauses fail (in particular, there is no pattern `_`) then a no-matching 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.

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 assigned 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.

This operator never makes backtracking. If there are several subterms matching the pattern, only the first match is considered. Note that the order of matching is left unspecified.

Pattern matching on goals

We can make pattern matching on goals using the following expression:

```
match goal with
| hyp1,1 , ... , hyp1,m1 | -cpattern1 => expr1
| hyp2,1 , ... , hyp2,m2 | -cpattern2 => expr2
...
| hypn,1 , ... , hypn,mn | -cpatternn => exprn
| _ => exprn+1
end
```

If each hypothesis pattern $hyp_{1,i}$, with $i = 1, \dots, m_1$ 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.

Error message: No matching clauses for match goal

No goal pattern can be used and, in particular, there is no `_ goal` pattern.

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`.

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.

As before, the following expression returns a term:

`fresh string`

It evaluates to an identifier unbound in the goal, which is obtained by padding *string* with a number if necessary. If no name is given, the prefix is H.

type of **term**

This tactic computes the type of *term*.

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`.

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_subproofn` 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

9.3 Tactic toplevel definitions

Basically, tactics toplevel definitions are made as follows:

```
Ltac ident ident1 ... identn := expr
```

This defines a new tactic that can be used in any tactic script or new tactic toplevel definition.

Remark: The preceding definition can equivalently be written:

```
Ltac ident := fun ident1 ... identn => expr
```

Recursive and mutual recursive function definitions are also possible with the syntax:

```
Ltac ident1 ident1,1 ... ident1,m1 := expr1
with ident2 ident2,1 ... ident2,m2 := expr2
...
with identn identn,1 ... identn,mn := exprn
```

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 657-669
> apply Rtrans.
> ^^^^^^^^^^^^^^^^^
Error: generated subgoal "R n ?17" has metavariables in it

```

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 ?5
subgoal 2 is:
R ?5 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 f : forest, P0 f -> P0 (cons t f)) ->
  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 `tree_forest_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 f : forest, P0 f -> P0 (cons t f)) ->
  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 <   | even0 : 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 $(n:\text{nat}) (\text{even } n) \rightarrow (Q \text{ } n)$.

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 <   | 0 => 0
Coq <   | S n0 => match n0 with
Coq <               | 0 => 0
Coq <               | S n' => S (div2 n')
Coq <           end
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.
```

You may now look at the type of div2_ind:

```

Coq < Check div2_ind.
div2_ind
  : forall Q0 : nat -> Prop,
    (forall n : nat, n = 0 -> Q0 0) ->
    (forall n n0 : nat, n = S n0 -> n0 = 0 -> Q0 1) ->
    (forall n n0 : nat,
      n = S n0 -> forall n' : nat, n0 = S n' -> Q0 n' -> Q0 (S (S n')) ->
      forall n : nat, Q0 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.
Coq < apply div2_ind; intros.
3 subgoals

  n : nat
  n0 : nat
  H_eq_ : n0 = 0
  =====
  div2 0 <= 0
subgoal 2 is:
  div2 1 <= 1
subgoal 3 is:
  div2 (S (S n')) <= S (S n')
Coq < auto with arith.
Coq < auto with arith.
Coq < simpl; auto with arith.
Coq < Qed.

```

Since `div2` is not mutually recursive, we can use directly the `functional induction` tactic instead of building the principle:

```

Coq < Reset div2_ind.
Coq < Lemma div2_le : forall n:nat, div2 n <= n.
Coq < intro n.
Coq < functional induction div2 n.
3 subgoals

  n : nat
  H_eq_ : n = 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 < auto with arith.
Coq < Qed.

```

remark: functional induction makes no use of an induction principle, so be warned that each time it is applied, a term mimicking the structure of `div2` (roughly the size of `div2_ind`) is built. Using `Functional Scheme` is generally faster and less memory consuming. On the other hand functional induction performs some extra simplifications that `Functional Scheme` does not, and as it is a tactic it can be used in tactic definitions.

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.

```
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 <   | empty => 0
Coq <   | cons t f' => (tree_size t + forest_size f')
Coq <   end.
```

The definition of principle of mutual induction following the recursive structure of `tree_size` is defined by the command:

```
Coq < Functional Scheme treeInd := Induction for tree_size
Coq < with tree_size forest_size.
```

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

```
Coq < Check treeInd.
treeInd
: forall (Q1 : forest -> Prop) (Q0 : tree -> Prop),
  (forall (t : tree) (A : A) (f : forest),
    t = node A f -> Q1 f -> Q0 (node A f)) ->
  (forall f : forest, f = empty -> Q1 empty) ->
  (forall (f : forest) (t : tree) (f' : forest),
    f = cons t f' -> Q0 t -> Q1 f' -> Q1 (cons t f')) ->
  forall t : tree, Q0 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 m0)` for certain `m0` and that `(Le n m0)`. 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 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.

Coq < Lemma ResAck0 :
Coq <   Ack 3 2 = 29.
1 subgoal

  =====
  Ack 3 2 = 29

Coq < autorewrite with base0 using try reflexivity.
Proof completed.

```

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.

```

10.7 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 $f : L \rightarrow A$. 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: $A \rightarrow L$.

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 (* constructor 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

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.

10.7.1 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 $A \wedge \neg A \rightarrow \text{False}$. 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

```

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.

10.7.2 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 19). 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)))))

```

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 19)

10.8 Using the tactical language

10.8.1 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 < 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

```

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).

10.8.2 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.

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

```

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

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.

```

```

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)

10.8.3 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 LJ^T* of Roy Dyckhoff ([49]), 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

```

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

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.

```

10.8.4 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, [38]). 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, [38]). 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 < 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 < Lemma isos_ex1 :
Coq <   forall A B:Set, A * unit * B = B * (unit * A).
Coq < Proof.
Coq < intros; IsoProve.
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 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)

Part III

User extensions

Chapter 11

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 11.1. 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 11.2.

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 11.3.

11.1 Notations

11.1.1 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 11.3). 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 11.1.7), explicit precedences and associativity rules have to be given.

11.1.2 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 $(A:\text{Prop})\text{True} \wedge A \vee A \vee \text{False}$ is ambiguous. To tell the COQ parser how to interpret the expression, a priority between the symbols \wedge and \vee 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 conjunction¹.

Similarly, an associativity is needed to decide whether $\text{True} \wedge \text{False} \wedge \text{False}$ defaults to $\text{True} \wedge (\text{False} \wedge \text{False})$ (right associativity) or to $(\text{True} \wedge \text{False}) \wedge \text{False}$ (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.

11.1.3 Complex notations

Notations can be made from arbitrarily 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).
```

One can also define notations for binders.

¹which are the levels effectively chosen in the current implementation of COQ

²COQ 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

```
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.

11.1.4 Simple factorization rules

COQ extensible parsing is performed by `Camlp4` 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.
```

11.1.5 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-or-else-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`.

11.1.6 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).
```

11.1.7 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.

11.1.8 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 11.1. 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 <   | 0 => m
Coq <   | S p => S (p+m)
Coq <   end
Coq < where "n + m" := (plus n m).
```

11.1.9 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 2.8.

<i>sentence</i>	<code>::=</code>	<code>Notation [Local] string := term [modifiers] [:scope] .</code> <code>Infix [Local] string := qualid [modifiers] [:scope] .</code> <code>Reserved Notation [Local] string [modifiers] .</code> <code>Inductive ind_body [decl_notation] with ... with ind_body [decl_notation].</code> <code>CoInductive ind_body [decl_notation] with ... with ind_body [decl_notation].</code> <code>Fixpoint fix_body [decl_notation] with ... with fix_body [decl_notation] .</code> <code>CoFixpoint cofix_body [decl_notation] with ... with cofix_body [decl_notation] .</code>
<i>decl_notation</i>	<code>::=</code>	<code>[where string := term [:scope]] .</code>
<i>modifiers</i>	<code>::=</code>	<code>ident , ... , ident at level natural</code> <code>ident , ... , ident at next level</code> <code>at level natural</code> <code>left associativity</code> <code>right associativity</code> <code>no associativity</code> <code>ident ident</code> <code>ident global</code> <code>ident bigint</code> <code>only parsing</code> <code>format string</code>

Figure 11.1: Syntax of the variants of Notation

11.1.10 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)

Coq < Locate "'exists' _ , _".
Notation          Scope
"'exists' x , p" := ex (fun x => p)
                  : type_scope
                  (default interpretation)
```

See also: Section 6.2.10.

11.1.11 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 $\dots (f\ t_1 \dots t_n) \dots$ can be used. Notice that \dots is part of the COQ syntax while \dots is just a meta-notation of this manual to denote a sequence of terms of arbitrary size.

This extra construction enclosed within \dots , let's call it t , must be one of the argument of an applicative term of the form $(f\ u_1 \dots u_n)$. The sequences $t_1 \dots t_n$ and $u_1 \dots u_n$ 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 " $x\ s \dots s\ y$ " where \dots 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 $(f\ u_1 \dots u_{i-1}\ [I]\ u_{i+1} \dots u_{j-1}\ [E]\ u_{j+1} \dots u_n)$ 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 " $x\ s \dots s\ y$ " 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 11.2).

11.1.12 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 3124-3125
> 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 " $\{ x : A \mid P \}$ " 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.

11.1.13 Summary

Syntax of notations The different syntactic variants of the command `Notation` are given on Figure 11.1. The optional `:scope` is described in the section 11.2.

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 `Notation Local` is used instead of `Notation`.

11.2 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. However, this is only made possible at the OBJECTIVE CAML level.

See Figure 11.1 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.

11.2.1 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. `Open Local Scope scope.`
2. `Close Local Scope scope.`

These variants are not exported to the modules that import the module where they occur, even if outside a section.

11.2.2 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.

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 16) 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 (plus x t) (Uplus (f nat (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.

11.2.3 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.

11.2.4 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`.

`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`.

11.2.5 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.

11.3 Abbreviations

An *abbreviation* is a name denoting a (presumably) more complex expression. An abbreviation is a special form of notation with no parameter and only one symbol which is an identifier. This identifier is given with no quotes around. Example:

```
Coq < Notation List := (list nat).
```

An abbreviation expects no precedence nor associativity, since it can always be put at the lower level of atomic expressions, and associativity is irrelevant. 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 like for an ordinary definition, and can be referred by partially 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, abbreviation can be bound to terms with holes (i.e. with “_”). The general syntax for abbreviations is

```
Notation [Local] 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.

Remark: compatibility Abbreviations are similar to the *syntactic definitions* available in versions of COQ prior to version 8.0, except that abbreviations are used for printing (unless the modifier (only parsing) is given) while syntactic definitions were not.

11.4 Tactic Notations

Tactic notations allow to customize the syntax of the tactics of the tactic language³. Tactic notations obey the following syntax

```

sentence      ::= Tactic Notation string [production_item ... production_item]
               := tactic .

production_item ::= string | tactic_argument_type (ident)
tactic_argument_type ::= ident | simple_intropattern | hyp
                      | reference | constr
                      | integer
                      | int_or_var | tactic |

```

A tactic notation `Tactic Notation string [production_item ... production_item] := tactic` extends the parser and pretty-printer of tactics with a new rule made of the juxtaposition of the head name of the tactic *string* and the list of its production items (in the syntax of production items, *string* stands for a terminal symbol and *tactic_argument_type(ident)* for non terminal entries. It then evaluates into the tactic expression *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	a tactic	

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`,

³Tactic notations are just a simplification of the Grammar `tactic simple_tactic` command that existed in versions prior to version 8.0.

`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.

Part IV

Practical tools

Chapter 12

The COQ commands

There are two COQ commands:

- `coqtop`: The COQ toplevel (interactive mode) ;
- `coqc` : The COQ compiler (batch compilation).

The options are (basically) the same for the two commands, and roughly described below. You can also look at the man pages of `coqtop` and `coqc` for more details.

12.1 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() ; ;`.

12.2 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.

12.3 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`.

12.4 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 `$COQTOP` for the directory of the sources. The latter is useful only for developers that are writing their own tactics and are using `coq_makefile` (see 13.3). If `$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.

12.5 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 *directory* to the searched directories when looking for a file.

`-R directory dirpath`

This maps the subdirectory structure of physical *directory* to logical *dirpath* and adds *directory* and its subdirectories to the searched directories when looking for a file.

`-top dirpath`

This sets the toplevel module name to *dirpath* instead of `Top`. Not valid for `coqc`.

`-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 13.4)

`-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.

`-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 launching `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.

Chapter 13

Utilities

The distribution provides utilities to simplify some tedious works beside proof development, tactics writing or documentation.

13.1 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 whose 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.

13.2 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.

13.3 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 `file1 ... filen`, and a generic target to produce a `.vo` file from the corresponding `.v` file (so you can do `make file.v 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.

13.4 Documenting COQ files with `coqdoc`

`coqdoc` is a documentation tool for the proof assistant COQ, similar to `javadoc` or `ocamldep`. 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.

13.4.1 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 %... $\dots$ ...% #...HTML...# *)
```

or

```
(** printing token $... $\dots$ ...$ #...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	\rightarrow	\leftarrow	\leftarrow	$*$	\times
\leq	\leq	\geq	\geq	\Rightarrow	\Rightarrow
\neq	\neq	\leftrightarrow	\leftrightarrow	\vdash	\vdash
\setminus	\vee	\wedge	\wedge	\sim	\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, $\rightarrow\sim$ is recognized as a single token, where COQ sees two tokens. It is the responsibility of the user to insert space between tokens *or* to give pretty-printing rules for the possible combinations, e.g.

```
(** printing  $\rightarrow\sim$  %\ensuremath{\rightarrow\lnot}% *)
```

Sections. Sections are introduced by 1 to 4 leading stars (i.e. at the beginning of the line). 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 `>>`. 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.

13.4.2 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.

\TeX macs output

To translate the input files to \TeX macs format, to be used by the \TeX macs 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 \TeX macs output.

`-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 file-name 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.

13.4.3 The coqdoc L^AT_EX style file

In case you choose to produce a document without the default L^AT_EX 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`.

13.5 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 HELM¹ and MoWGLI² projects mainly at the University of Bologna and partly at INRIA-Sophia Antipolis.

13.5.1 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 13.3), 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.

¹Hypertextual Electronic Library of Mathematics

²Mathematics on the Web, Get it by Logic and Interfaces

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.

13.5.2 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 $ident_1 ident_n . ident$ is exported to files in the subdirectory $ident_1 / . . . / ident_n$ of the directory bound to the environment variable `COQ_XML_LIBRARY_ROOT`.

13.5.3 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 13.4). Especially, they have to be enclosed within `(** and *)` to be exported.

For each inductive definition of name $ident_1 ident_n . ident$, a file named $ident . ind . xml$ is created in the subdirectory $ident_1 / . . . / ident_n$ 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 $ident_1 ident_n . ident$, files named $ident . con . body . xml$ and $ident . con . xml$ are created in the subdirectory $ident_1 / . . . / ident_n$. They respectively contain the body and the type of the definition.

For each global assumption of base name $ident_1 . ident_2 ident_n . ident$, a file named $ident . con . xml$ is created in the subdirectory $ident_1 / . . . / ident_n$. It contains the type of the global assumption.

For each local assumption or definition of base name $ident$ located in sections $ident'_1, \dots, ident'_p$ of the module $ident_1 . ident_2 ident_n . ident$, a file named $ident . var . xml$ is created in the subdirectory $ident_1 / . . . / ident_n / ident'_1 / . . . / ident'_p$. 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.

13.5.4 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).

³or `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.

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 3rd 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 3rd condition leads to XML file that are two times as big as the ones exported applying the 3rd condition.

13.5.5 Interactive exportation commands

There are also commands to be used interactively in `coqtop`.

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*.

13.5.6 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.

13.5.7 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.2.3). 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.

13.6 Embedded COQ phrases inside L^AT_EX documents

When writing a documentation about a proof development, one may want to insert COQ phrases inside a L^AT_EX document, possibly together with the corresponding answers of the system. We provide a mechanical way to process such COQ phrases embedded in L^AT_EX 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.

13.7 COQ and GNU EMACS

13.7.1 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.

13.7.2 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`.

13.8 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.

13.9 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 14

CoQ Integrated Development Environment

The CoQ 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 12, the ones having obviously no meaning for CoQIDE being ignored.

A sample CoQIDE main screen, while navigating into a file `Fermat.v`, is shown on Figure 14.1. 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.

14.1 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.

14.2 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 14.1, 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

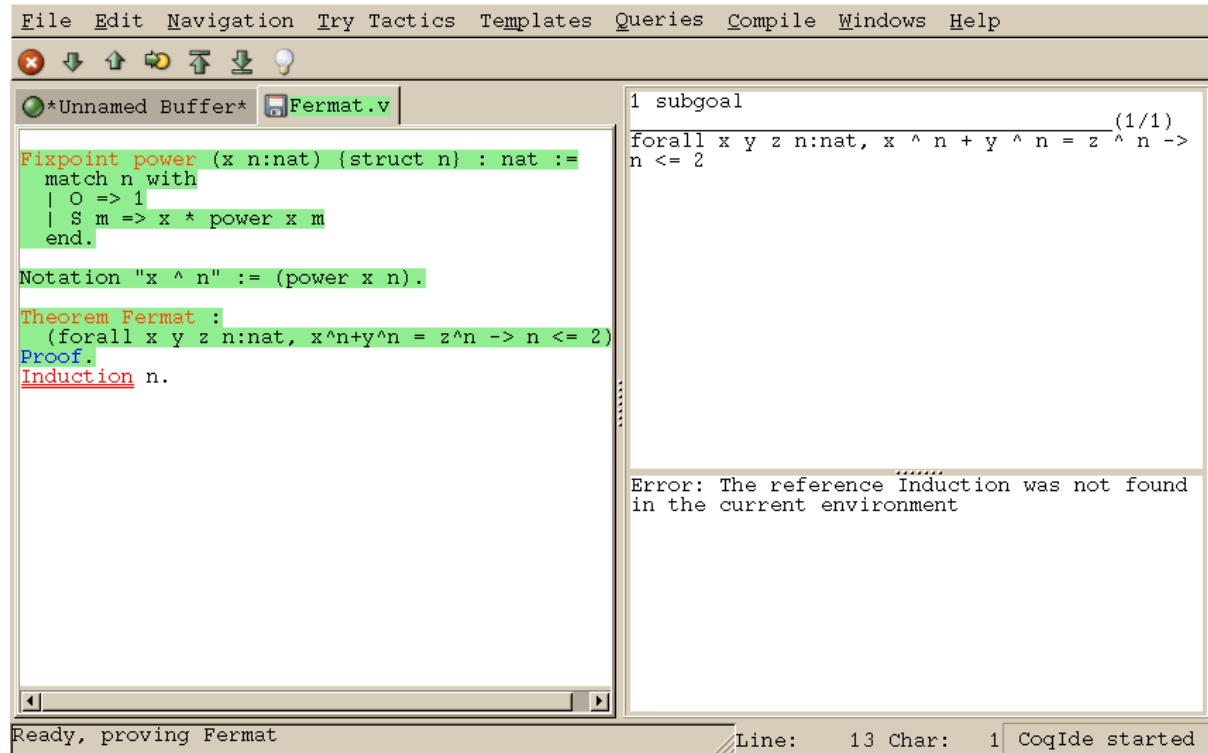


Figure 14.1: COQIDE main screen

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.

14.3 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 on the goal or the considered hypothesis. This is the “contextual menu on goals” feature, that may be disabled in the preferences if undesirable.

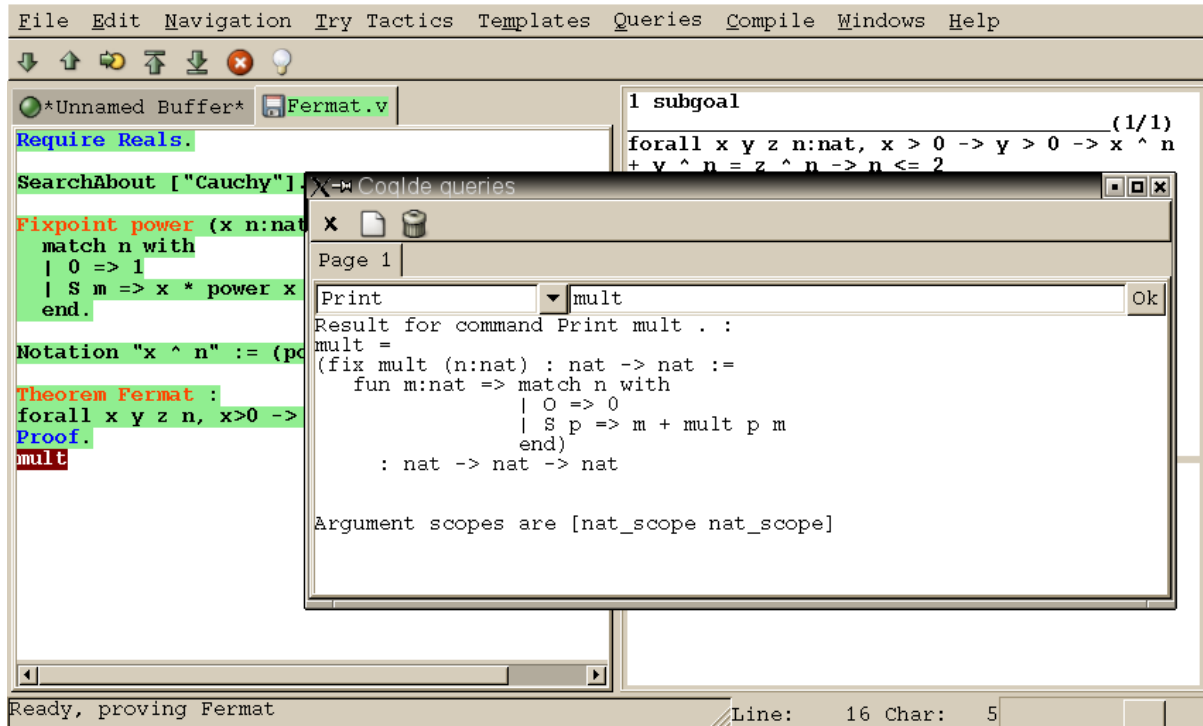


Figure 14.2: COQIDE: the query window

14.4 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.

14.5 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 14.2 displays the query window after selection of the word “mult” in the script windows, and pressing `F4` to print its definition.

14.6 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`.

14.7 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 14.8.3

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 14.3.

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 exists in the COQ library, you may copy this file into your home directory, and edit it using any text editor, COQIDE itself for example.

14.8 Using unicode symbols

COQIDE supports unicode character encoding in its text windows, consequently a large set of symbols is available for notations.

14.8.1 Displaying unicode symbols

You just need to define suitable notations as described in Chapter 11. For example, to use the mathematical symbols \forall and \exists , 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.

14.8.2 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 \forall 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 \forall and \exists respectively, you may add

```
bind "F11" "insert-at-cursor" ("∀")
bind "F12" "insert-at-cursor" ("∃")
```

to your binding "text" section in `.coqiderc-gtk2rc`.

14.8.3 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.

14.9 Building a custom COQIDE with user ML code

You can do this as described in Section 13.1 for a custom coq text toplevel, simply by adding option `-ide` to `coqmktop`, that is something like

```
coqmktop -ide -byte m1.cmo ... m_n.cmo
```

or

```
coqmktop -ide -opt m1.cmx ... m_n.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 these 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.

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.

Contents

Practical use of the XML exportation tool	224
Reflection of the logical structure into the file system	225
What is exported?	225
Inner types	225
Interactive exportation commands	226
Applications: rendering, searching and publishing	226
Technical informations	226
Extended pattern-matching	243
Patterns	243
About patterns of parametric types	246
Matching objects of dependent types	246
Understanding dependencies in patterns	247
When the elimination predicate must be provided	247
Using pattern matching to write proofs	248
Pattern-matching on inductive objects involving local definitions	249
Pattern-matching and coercions	250
When does the expansion strategy fail ?	250

Implicit Coercions	253
General Presentation	253
Classes	253
Coercions	254
Identity Coercions	254
Inheritance Graph	254
Declaration of Coercions	255
Coercion <i>qualid</i> : <i>class</i> ₁ \rightarrow <i>class</i> ₂ .	255
Identity Coercion <i>ident</i> : <i>class</i> ₁ \rightarrow <i>class</i> ₂ .	256
Displaying Available Coercions	257
Print Classes.	257
Print Coercions.	257
Print Graph.	257
Print Coercion Paths <i>class</i> ₁ <i>class</i> ₂ .	257
Activating the Printing of Coercions	257
Set Printing Coercions.	257
Set Printing Coercion <i>qualid</i> .	257
Classes as Records	257
Coercions and Sections	258
Examples	258
 Omega: a solver of quantifier-free problems in Presburger Arithmetic	 263
Description of omega	263
Arithmetical goals recognized by omega	263
Messages from omega	264
Technical data	265
Overview of the tactic	265
Overview of the <i>OMEGA</i> decision procedure	265
Bugs	265
 Extraction of programs in Objective Caml and Haskell	 267
Generating ML code	267
Extraction options	268
Setting the target language	268
Inlining and optimizations	268
Realizing axioms	269
Differences between COQ and ML type systems	271
Some examples	271
A detailed example: Euclidean division	272
Another detailed example: Heapsort	273
The Standard Library	276
Extraction's horror museum	276
Users' Contributions	277
 The <i>ring</i> tactic	 279
What does this tactic?	279
The variables map	279
Is it automatic?	280
Concrete usage in COQ	280
Add a ring structure	281

Presentation of the Addendum	241
How does it work?	283
History of ring	284
Discussion	284
The <code>setoid_replace</code> tactic	287
Description of <code>setoid_replace</code>	287
Adding new setoid or morphisms	287
Adding new morphisms	288
The tactic itself	288

Chapter 15

Extended pattern-matching

Cristina Cornes

This section describes the full form of pattern-matching in COQ terms.

15.1 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 variables, it is called *simple*: it is the kind of pattern recognized by the basic version of `match`. If a pattern is not simple we call it *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 list of patterns separated with commas is also considered as a pattern and is called *multiple pattern*.

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. An easy way to see the result of the expansion is 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 \Rightarrow (*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 <   | 0 => 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

```

Using multiple patterns in the definition 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

```

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

```

We can also use “as patterns” 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

```

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

```

This is compiled into:

```

Coq < Print even.
even =
fix even (n : nat) : bool :=
  match n with

```

```

| 0 => true
| S n0 => match n0 with
| 0 => false
| 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 `leq` 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 leq (n m:nat) {struct m} : bool :=
Coq <   match n, m with
Coq <   | 0, x => true
Coq <   | x, 0 => false
Coq <   | S n, S m => leq n m
Coq <   end.
leq is recursively defined

```

Note that the first and the second multiple pattern superpose because the couple of values `0 0` 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 *i*th row if and only if it is not matched by some pattern of a previous row. Thus in the example, `0 0` is matched by the first pattern, and so `(leq 0 0)` yields `true`.

Another way to write this function is:

```

Coq < Reset leq.

Coq < Fixpoint leq (n m:nat) {struct m} : bool :=
Coq <   match n, m with
Coq <   | 0, x => true
Coq <   | S n, S m => leq n m
Coq <   | _, _ => false
Coq <   end.
leq is recursively defined

```

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 <   | 0 => true
Coq <   | S _ => false
Coq <   | x => true
Coq <   end).
Coq < Coq < Toplevel input, characters 1248-1257
>   / x => true
>   ^^^^^^^^^
Error: This clause is redundant

```

15.2 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 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 1700-1711
>       | cons A _ l' => l'
>       ^^^^^^^^^^^
Error: The constructor cons expects 2 arguments.
```

15.3 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
```

15.3.1 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.

15.3.2 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
```

The elimination predicate is `fun (n:nat) (l:listn n) => listn (n+m)`. In general if m has type $(I\ q_1 \dots q_r\ t_1 \dots t_s)$ where $q_1 \dots q_r$ are parameters, the elimination predicate should be of the form `fun y1...ys x:(I q1...qr y1...ys) => P`.

In the concrete syntax, it should be written :

$$\text{match } m \text{ as } x \text{ in } (I _ \dots _ y_1 \dots y_s) \text{ return } Q \text{ with } \dots \text{ end}$$

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

```

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 3232-3233
>   / 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 (?31 + ?32)"

```

15.4 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 is 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 <   | 0 => niln
Coq <   | S n => consn n 0 (buildlist n)
Coq <   end.
buildlist is recursively defined

```

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 $(n, m : \text{nat}) \rightarrow (\text{LE } n \text{ } m) \rightarrow (\text{LE } m \text{ } n)$:

```
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 <   | 0, x => or_introl (LE x 0) (LEO x)
Coq <   | x, 0 => 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
```

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.

15.5 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
```

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
```

Remark: for a given matching clause, either none of the local definitions or all of them can be caught.

15.6 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 0 => 0
Coq <       | _  => 0
Coq <       end).
fun x : I =>
match x with
| C1 _ => 0
| C2 i =>
    match i with
    | C1 n => match n with
    | 0 => 0
    | S _ => 0
    end
    | C2 _ => 0
    end
end
: I -> nat
```

15.7 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 0 x) => e1` and `(cons n _ x) => e2` instead of `(cons n 0 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 incompatibility or the problem involves dependencies

There is a type mismatch between the different branches

Then the user should provide an elimination predicate.

Chapter 16

Implicit Coercions

Amokrane Saïbi

16.1 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.

16.2 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 16.1.

<i>class</i>	<code>::=</code>	<i>qualid</i>
		<code>Sortclass</code>
		<code>Funclass</code>

Figure 16.1: Syntax of classes

16.3 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 $\text{forall } (x_1 : A_1) .. (x_n : A_n) (y : C \ x_1 .. x_n), D \ u_1 .. u_m$ where m is the number of parameters of D .
- D is `Funclass`, then the type of f must have the form $\text{forall } (x_1 : A_1) .. (x_n : A_n) (y : C \ x_1 .. x_n) (x : A), B$.
- D is `Sortclass`, then the type of f must have the form $\text{forall } (x_1 : A_1) .. (x_n : A_n) (y : C \ x_1 .. x_n), s$ with s a sort.

We then write $f : C \multimap D$. 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 .

16.4 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 $f : \text{forall } (x_1 : T_1) .. (x_k : T_k) (y : C \ u_1 .. u_n), D \ v_1 .. v_m$ 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 :

$$C' := \text{fun } (x_1 : T_1) .. (x_k : T_k) => C \ u_1 .. u_n$$

We then define an *identity coercion* between C' and C :

$$\text{Id_C'_C} := \text{fun } (x_1 : T_1) .. (x_k : T_k) (y : C' \ x_1 .. x_k) => (y : C \ u_1 .. u_n)$$

We can now declare f as coercion from C' to D , since we can “cast” its type as $\text{forall } (x_1 : T_1) .. (x_k : T_k) (y : C' \ x_1 .. x_k), D \ v_1 .. v_m$.

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 $\text{Id_C'_C} \ t_1 .. t_k \ t$ 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 .

16.5 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 $[f_1; \dots; f_k] : C \multimap D$ 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.

16.6 Declaration of Coercions

16.6.1 Coercion *qualid* : $class_1 \multimap class_2$.

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: $[f_1^1; \dots; f_{n_1}^1] : C_1 \multimap D_1$
 \dots
 $[f_1^m; \dots; f_{n_m}^m] : C_m \multimap D_m$

Variants:

1. Coercion Local *qualid* : $class_1 \multimap class_2$.
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 its 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 its source and its target.

4. Coercion Local *ident* := *term*
This defines *ident* just like Local *ident* := *term*, and then declares *ident* as a coercion between its 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.

16.6.2 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*}) => (*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* : *class*₁ *x*₁ .. *x*_{*n*}), *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. Identity Coercion Local *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.

16.7 Displaying Available Coercions

16.7.1 Print Classes.

Print the list of declared classes in the current context.

16.7.2 Print Coercions.

Print the list of declared coercions in the current context.

16.7.3 Print Graph.

Print the list of valid coercion paths in the current context.

16.7.4 Print Coercion Paths `class1 class2.`

Print the list of valid coercion paths from `class1` to `class2`.

16.8 Activating the Printing of Coercions

16.8.1 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.

16.8.2 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.

16.9 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:

$$\begin{array}{l} \text{Record } [>] \text{ ident binderlet : sort := [ident}_0\text{] } \{ \\ \quad \text{ident}_1 \text{ [:|:>] term}_1 \text{ ;} \\ \quad \dots \\ \quad \text{ident}_n \text{ [:|:>] term}_n \text{ } \} . \end{array}$$

The identifier `ident` is the name of the defined record and `sort` is its type. The identifier `ident0` is the name of its constructor. The identifiers `ident1`, ..., `identn` are the names of its fields and `term1`, ..., `termn` their respective types. The alternative `[:|:>]` is “:” or “:>”. If `identi :> termi`, then `identi` is automatically declared as coercion from `ident` to the class of `termi`. Remark that `identi` always verifies the uniform inheritance condition. If the optional “>” before `ident` is present, then `ident0` (or the default name `Build_ident` if `ident0` is omitted) is automatically declared as a coercion from the class of `termn` to `ident` (this may fail if the uniform inheritance condition is not satisfied).

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

16.10 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.

16.11 Examples

There are three situations:

- $f\ a$ is ill-typed where $f : \text{forall } x : A, B$ 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
```

```
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 $t : \text{forall } x : A, B$ towards $\text{forall } x : A', B'$, 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 $\text{fun } x : A \Rightarrow t$, universal quantification $\text{forall } x : A, B$, global variables and parameters of (co-)inductive definitions and functions. In $\text{forall } x : A, B$, 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 `Funcclass` 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.
[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 17

Omega: a solver of quantifier-free problems in Presburger Arithmetic

Pierre Crégut

17.1 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.

17.1.1 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`.

17.1.2 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 existential 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*

17.2 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.
```


17.3 Technical data

17.3.1 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 \mathbb{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.

17.3.2 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.

17.4 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 18

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).

18.1 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 qualid1 ... qualidn .`

Recursive extraction of all the globals (or modules) *qualid*₁ ... *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 chosen ML language to fulfill 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.

18.2 Extraction options

18.2.1 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.

18.2.2 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 laziness. 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 simplifications 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 qualid1 ... qualidn.`

`Extraction NoInline qualid1 ... qualidn.`

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 explicitly 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.

18.2.3 Realizing axioms

Extraction will fail if it encounters an informative axiom not realized (see section 18.2.3). 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 axiom.

Of course, it is the responsibility 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 string1 ...stringn => string .`

The number of type variable given 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 inconsistent 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" ].
```

18.3 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 [84].

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).

18.4 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.

18.4.1 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 nat =
  | O
  | S of nat
type sumbool =
  | Left
  | Right
(** val minus : nat -> nat -> nat **)
let rec minus n m =
  match n with
  | O -> O
  | S k -> (match m with
            | O -> S k
            | 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 n1 -> le_lt_dec n0 n1)
(** val le_gt_dec : nat -> nat -> sumbool **)
let le_gt_dec n m =
  le_lt_dec n m
type diveucl =
  | Divex of nat * nat
(** val eucl_dev : nat -> nat -> diveucl **)
let rec eucl_dev b a =
  match le_gt_dec b a with
  | Left -> let Divex (x, x0) = eucl_dev b (minus a b) in Divex ((S x), x0)
  | Right -> Divex (O, 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 = 0 | 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 -> 0 | n -> S (i2n (n-1));;
val i2n : int -> nat = <fun>
# let rec n2i = function 0 -> 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
```

18.4.2 Another detailed example: Heapsort

The file `Heap.v` contains the proof of an efficient list sorting algorithm described by Bjerner. It is an adaptation of the well-known *heapsort* algorithm to functional languages. The main function is `treesort`, whose type is shown below:

```
Coq < Require Import Heap.
Coq < Check treesort.
treesort
  : forall (A : Set) (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 =
  | 0
  | 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 = 0 | 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.

18.4.3 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`).

The result of this extraction of the Standard Library can be browsed at URL

<http://www.lri.fr/~letouzey/extraction>.

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.

18.4.4 Extraction's horror museum

Some pathological examples of extraction are grouped in the file

`contrib/extraction/test_extraction.v`

of the sources of COQ.

18.4.5 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 [84].

Chapter 19

The `ring` tactic

Patrick Loiseleur and Samuel Boutin

This chapter presents the `ring` tactic.

19.1 What does this tactic?

`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, \dots and constants by application of \oplus and \otimes .

Let an *ordered product* be a product of variables $V_{i_1} \otimes \dots \otimes V_{i_n}$ verifying $i_1 \leq i_2 \leq \dots \leq i_n$. Let a *monomial* be the product of a constant (possibly equal to 1, in which case we omit it) 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. 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)

19.2 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 ↦ if b then (4) else (f (3))
1 ↦ (f (5))
2 ↦ 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)))
```

19.3 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.

19.4 Concrete usage in Coq

Under a session launched by `coqtop` or `coqtop -full`, load the `ring` files with the command:

```
Require Ring.
```

It does not work under `coqtop -opt` because the compiled ML objects used by the tactic are not linked in this binary image, and dynamic loading of native code is not possible in OBJECTIVE CAML.

In order to use `ring` on naturals, load `ArithRing` instead; for binary integers, load the module `ZArithRing`.

Then, to normalize the terms $term_1, \dots, term_n$ in the current subgoal, use the tactic:

```
ring term1 ... termn
```

Then the tactic guesses the type of given terms, the ring theory to use, the variables map, and replace each term with its normal form. The variables map is common to all terms

Warning: `ring term1; ring term2` is not equivalent to `ring term1 term2`. 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.

Error messages:

1. All terms must have the same type
2. Don't know what to do with this goal
3. No Declared Ring Theory for *term*.

Use `Add [Semi] Ring` to declare it

That happens when all terms have the same type *term*, but there is no declared ring theory for this set. See below.

Variants:

1. ring

That works if the current goal is an equality between two polynomials. It will normalize both sides of the equality, solve it if the normal forms are equal and in other cases try to simplify the equality using `congr_eqT` and `refl_equal` to reduce $x + y = x + z$ to $y = z$ and $x * z = x * y$ to $y = z$.

Error message: This goal is not an equality

19.5 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 \mathbb{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 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 $A \rightarrow A \rightarrow A$, $Amult$ is a term of type $A \rightarrow A \rightarrow A$, $Aone$ is a term of type A , $Azero$ is a term of type A , $Ainv$ is a term of type $A \rightarrow A$, Aeq is a term of type $A \rightarrow A \rightarrow \text{bool}$, T is a term of type $(\text{Ring_Theory } A Aplus Amult Aone Azero Ainv Aeq)$. The arguments $c1 \dots cn$, 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 $c1 \dots cn$ 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 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 $(\text{Semi_Ring_Theory } A Aplus Amult Aone Azero Aeq)$.

2. `Add 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 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 that 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`.

19.6 How does it work?

The code of `ring` is a good example of tactic written using *reflection* (or *internalization*, it is synonymous). 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_normalize.v`. Here a type for polynomials is defined:

```
Inductive Type polynomial :=
  Pvar : idx -> polynomial
| Pconst : A -> polynomial
| Pplus : polynomial -> polynomial -> polynomial
| Pmult : polynomial -> polynomial -> polynomial
| Popp : polynomial -> polynomial.
```

There is also a type to represent variables maps, and an interpretation function, that maps a variables map and a polynomial to an element of the concrete ring:

```
Definition polynomial_simplify := [...]
Definition interp : (varmap A) -> (polynomial A) -> A := [...]
```

A function to normalize polynomials is defined, and the big theorem is its correctness w.r.t interpretation, that is:

```
Theorem polynomial_simplify_correct : forall (v:(varmap A))(p:polynomial)
  (interp v (polynomial_simplify p))
  =(interp v p).
```

(The actual code is slightly more complex: for efficiency, there is a special datatype to represent normalized polynomials, i.e. “canonical sums”. But the idea is still the same).

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 `(interp v ap)`. Then we replace it by `(interp v (polynomial_simplify 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:

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 a non-sense to replace them by tactics using reflection.

Another argument against the reflection is that `COQ`, as a programming language, has many nice features, like dependent types, but is very far from the speed and the expressive power of `OBJECTIVE CAML`. Wait a minute! With `COQ` it is possible to extract ML code from `pCIC` terms, right? So, why not to link the extracted code with `COQ` to inherit the benefits of the reflection and the speed of ML tactics? That is called *total reflection*, and is still an active research subject. With these technologies it will become possible to bootstrap the type-checker of `pCIC`, but there is still some work to achieve that goal.

Another brilliant idea from Benjamin Werner: reflection could be used to couple a 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 replay the trace in `Coq`, and apply the correction lemma. So internalization seems to be the best way to import ... external proofs!

Chapter 20

The `setoid_replace` tactic

Clément Renard

This chapter presents the `setoid_replace` tactic.

20.1 Description of `setoid_replace`

Working on user-defined structures in COQ is not very easy if Leibniz equality does not denote the intended equality. For example using lists to denote finite sets drive to difficulties since two non convertible terms can denote the same set.

We present here a COQ module, `setoid_replace`, which allows to structure and automate some parts of the work. In particular, if everything has been registered a simple tactic can do replacement just as if the two terms were equal.

20.2 Adding new setoid or morphisms

Under the toplevel load the `setoid_replace` files with the command:

```
Coq < Require Setoid.
```

A setoid is just a type `A` and an equivalence relation on `A`.

The specification of a setoid can be found in the file

```
theories/Setoids/Setoid.v
```

It looks like :

```
Section Setoid.
```

```
Variable A : Type.
```

```
Variable Aeq : A -> A -> Prop.
```

```
Record Setoid_Theory : Prop :=
```

```
{ Seq_refl : (x:A) (Aeq x x);
```

```
  Seq_sym : (x,y:A) (Aeq x y) -> (Aeq y x);
```

```
  Seq_trans : (x,y,z:A) (Aeq x y) -> (Aeq y z) -> (Aeq x z)
```

```
}.
```

To define a setoid structure on A , you must provide a relation Aeq on A and prove that Aeq is an equivalence relation. That is, you have to define an object of type $(Setoid_Theory\ A\ Aeq)$.

Finally to register a setoid the syntax is:

```
Add Setoid A Aeq ST
```

where Aeq is a term of type $A \rightarrow A \rightarrow Prop$ and ST is a term of type $(Setoid_Theory\ A\ Aeq)$.

Error messages:

1. Not a valid setoid theory.
That happens when the typing condition does not hold.
2. A `Setoid Theory` is already declared for A .
That happens when you try to declare a second setoid theory for the same type.

Currently, only one setoid structure may be declared for a given type. This allows automatic detection of the theory used to achieve the replacement.

The table of setoid theories is compatible with the COQ sectioning mechanism. If you declare a setoid inside a section, the declaration will be thrown away when closing the section. And when you load a compiled file, all the `Add Setoid` commands of this file that are not inside a section will be loaded.

Warning: Only the setoid on `Prop` is loaded by default with the `setoid_replace` module. The equivalence relation used is `iff` *i.e.* the logical equivalence.

20.3 Adding new morphisms

A morphism is nothing else than a function compatible with the equivalence relation. You can only replace a term by an equivalent in position of argument of a morphism. That's why each morphism has to be declared to the system, which will ask you to prove the accurate compatibility lemma.

The syntax is the following :

```
Add Morphism f : ident
```

where f is the name of a term which type is a non dependent product (the term you want to declare as a morphism) and *ident* is a new identifier which will denote the compatibility lemma.

Error messages:

1. The term *term* is already declared as a morphism
2. The term *term* is not a product
3. The term *term* should not be a dependent product

The compatibility lemma generated depends on the setoids already declared.

20.4 The tactic itself

After having registered all the setoids and morphisms you need, you can use the tactic called `setoid_replace`. The syntax is

```
setoid_replace term1 with term2
```

The effect is similar to the one of `replace`.

You also have a tactic called `setoid_rewrite` which is the equivalent of `rewrite` for setoids. The syntax is

```
setoid_rewrite term
```

Variants:

1. `setoid_rewrite -> term`
2. `setoid_rewrite <- term`

The arrow tells the system in which direction the rewriting has to be done. Moreover, you can use `rewrite` for setoid rewriting. In that case the system will check if the term you give is an equality or a setoid equivalence and do the appropriate work.

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 pp 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] Yves bertot and Pierre Castéran. *Coq'Art*. Springer-Verlag, 2004. To appear.
- [14] E. Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, 1967.
- [15] S. Boutin. Certification d'un compilateur ML en Coq. Master's thesis, Université Paris 7, September 1992.
- [16] S. Boutin. *Réflexions sur les quotients*. thèse d'université, Paris 7, April 1997.

- [17] 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.
- [18] R.S. Boyer and J.S. Moore. *A computational logic*. ACM Monograph. Academic Press, 1979.
- [19] Laurent Chicli, Loïc Pottier, and Carlos Simpson. Mathematical quotients and quotient types in coq. In *TYPES'02*, volume 2646 of *Lecture Notes in Computer Science*, Berg en Dal, The Netherlands, 2003. Springer-Verlag.
- [20] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.
- [21] Th. Coquand. *Une Théorie des Constructions*. PhD thesis, Université Paris 7, January 1985.
- [22] Th. Coquand. An Analysis of Girard's Paradox. In *Symposium on Logic in Computer Science*, Cambridge, MA, 1986. IEEE Computer Society Press.
- [23] 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 [57].
- [24] Th. Coquand. A New Paradox in Type Theory. In *Proceedings 9th Int. Congress of Logic, Methodology and Philosophy of Science*, August 1991.
- [25] Th. Coquand. Pattern Matching with Dependent Types. In Nordström et al. [93].
- [26] Th. Coquand. Infinite Objects in Type Theory. In Barendregt and Nipkow [7].
- [27] 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.
- [28] 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.
- [29] Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.
- [30] 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.
- [31] J. Courant. Explicitation de preuves par récurrence implicite. Master's thesis, DEA d'Informatique, ENS Lyon, September 1994.
- [32] 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.
- [33] 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.
- [34] D. de Rauglaudre. Camlp4 version 1.07.2. In Camlp4 distribution, 1998.
- [35] 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.

- [36] 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.
- [37] 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.
- [38] 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.
- [39] G. Dowek. Naming and scoping in a mathematical vernacular. Research Report 1283, INRIA, 1990.
- [40] G. Dowek. *Démonstration automatique dans le Calcul des Constructions*. PhD thesis, Université Paris 7, December 1991.
- [41] 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.
- [42] 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.
- [43] G. Dowek. A Complete Proof Synthesis Method for the Cube of Type Systems. *Journal Logic Computation*, 3(3):287–315, June 1993.
- [44] G. Dowek. The undecidability of pattern matching in calculi where primitive recursive functions are representable. *Theoretical Computer Science*, 107(2):349–356, 1993.
- [45] G. Dowek. Third order matching is decidable. *Annals of Pure and Applied Logic*, 69:135–155, 1994.
- [46] G. Dowek. Lambda-calculus, combinators and the comprehension schema. In *Proceedings of the second international conference on typed lambda calculus and applications*, 1995.
- [47] 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.
- [48] 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.
- [49] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3), September 1992.
- [50] 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.
- [51] J.-C. Filliâtre. A decision procedure for direct predicate calculus. Research report 96–25, LIP-ENS-Lyon, 1995.

- [52] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université Paris-Sud, July 1999.
- [53] J.-C. Filliâtre. Formal Proof of a Program: Find. Submitted to *Science of Computer Programming*, January 2000.
- [54] 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 [52]].
- [55] J.-C. Filliâtre and N. Magaud. Certification of sorting algorithms in the system COQ. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.
- [56] E. Fleury. Implantation des algorithmes de Floyd et de Dijkstra dans le Calcul des Constructions. Rapport de Stage, July 1990.
- [57] Projet Formel. The Calculus of Constructions. Documentation and user’s guide, Version 4.10. Technical Report 110, INRIA, 1989.
- [58] Jean-Baptiste-Joseph Fourier. *Fourier’s method to solve linear inequations/equations systems*. Gauthier-Villars, 1890.
- [59] 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.
- [60] 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.
- [61] E. Giménez. A tutorial on recursive types in coq. Technical report, INRIA, March 1998.
- [62] 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.
- [63] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur*. PhD thesis, Université Paris 7, 1972.
- [64] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989.
- [65] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Centre., 1995.
- [66] D. Hirschhoff. Écriture d’une tactique arithmétique pour le système COQ. Master’s thesis, DEA IARFA, Ecole des Ponts et Chaussées, Paris, September 1994.
- [67] 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.
- [68] 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.

- [69] 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.
- [70] 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 [69].
- [71] G. Huet, editor. *Logical Foundations of Functional Programming*. The UT Year of Programming Series. Addison-Wesley, 1989.
- [72] 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 [57].
- [73] 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.
- [74] G. Huet. Residual theory in λ -calculus: a formal development. *J. Functional Programming*, 4,3:371–394, 1994.
- [75] G. Huet and J.-J. Lévy. Call by need computations in non-ambiguous 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.
- [76] G. Huet and G. Plotkin, editors. *Logical Frameworks*. Cambridge University Press, 1991.
- [77] G. Huet and G. Plotkin, editors. *Logical Environments*. Cambridge University Press, 1992.
- [78] J. Ketonen and R. Weyhrauch. A decidable fragment of Predicate Calculus. *Theoretical Computer Science*, 32:297–307, 1984.
- [79] S.C. Kleene. *Introduction to Metamathematics*. Bibliotheca Mathematica. North-Holland, 1952.
- [80] J.-L. Krivine. *Lambda-calcul types et modèles*. Etudes et recherche en informatique. Masson, 1990.
- [81] A. Laville. Comparison of priority rules in pattern matching and term rewriting. *Journal of Symbolic Computation*, 11:321–347, 1991.
- [82] 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.
- [83] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.
- [84] P. Letouzey. A new extraction for coq. In *Proceedings of the TYPES'2002 workshop*, 2002. to appear.
- [85] L. Puel and A. Suárez. Compiling Pattern Matching by Term Decomposition. In *Conference Lisp and Functional Programming*, ACM. Springer-Verlag, 1990.
- [86] P. Manoury. A User's Friendly Syntax to Define Recursive Functions as Typed λ -Terms. In *Types for Proofs and Programs, TYPES'94*, volume 996 of *LNCS*, June 1994.

- [87] P. Manoury and M. Simonot. Automatizing termination proof of recursively defined function. *TCS*, To appear.
- [88] L. Maranget. Two Techniques for Compiling Lazy Pattern Matching. Technical Report 2385, INRIA, 1994.
- [89] C. Muñoz. Démonstration automatique dans la logique propositionnelle intuitionniste. Master's thesis, DEA d'Informatique Fondamentale, Université Paris 7, September 1994.
- [90] 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.
- [91] B. Nordström. Terminating general recursion. *BIT*, 28, 1988.
- [92] 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.
- [93] 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.
- [94] P. Odifreddi, editor. *Logic and Computer Science*. Academic Press, 1990.
- [95] P. Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, 1984.
- [96] 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].
- [97] C. Parent. *Synthèse de preuves de programmes dans le Calcul des Constructions Inductives*. PhD thesis, Ecole Normale Supérieure de Lyon, 1995.
- [98] 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.
- [99] M. Parigot. Recursive Programming with Proofs. *Theoretical Computer Science*, 94(2):335–356, 1992.
- [100] 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.
- [101] 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.
- [102] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Université Paris 7, January 1989.
- [103] 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.
- [104] C. Paulin-Mohring. *Le système Coq. Thèse d'habilitation*. ENS Lyon, January 1997.
- [105] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

- [106] K.V. Prasad. Programming with broadcasts. In *Proceedings of CONCUR'93*, volume 715 of *LNCS*. Springer-Verlag, 1993.
- [107] J. Rouyer. Développement de l'Algorithme d'Unification dans le Calcul des Constructions. Technical Report 1795, INRIA, November 1992.
- [108] A. Saïbi. Axiomatization of a lambda-calculus with explicit-substitutions in the Coq System. Technical Report 2345, INRIA, December 1994.
- [109] H. Saidi. Résolution d'équations dans le système λ de gödel. Master's thesis, DEA d'Informatique Fondamentale, Université Paris 7, September 1994.
- [110] T. Streicher. Semantical investigations into intensional type theory, 1993. Habilitationsschrift, LMU Munchen.
- [111] Lemme Team. Pcoq a graphical user-interface for Coq. <http://www-sop.inria.fr/lemme/pcoq/>.
- [112] The Coq Development Team. The Coq Proof Assistant Reference Manual Version 7.2. Technical Report 255, INRIA, 2002.
- [113] D. Terrasse. Traduction de TYPOL en COQ. Application à Mini ML. Master's thesis, IARFA, September 1992.
- [114] L. Théry, Y. Bertot, and G. Kahn. Real theorem provers deserve real user-interfaces. Research Report 1684, INRIA Sophia, May 1992.
- [115] 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.
- [116] P. Wadler. Efficient compilation of pattern matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- [117] P. Weis and X. Leroy. *Le langage Caml*. InterEditions, 1993.
- [118] B. Werner. *Une théorie des constructions inductives*. Thèse de doctorat, Université Paris 7, 1994.

Global Index

- `||`, 170
- `*`, 71, 76
- `+`, 71, 76
- `-`, 76
- `/`, 76
- `:`, 169
- `:[...|...|...]`, 169
- `<`, 76
- `<=`, 76
- `>`, 76
- `>=`, 76
- `?`, 128
- `?=`, 76
- `%`, 205
- `&`, 72
- `_`, 30
- `{A}+{B}`, 72
- `{x:A & (P x)}`, 72
- `{x:A | (P x)}`, 71
- `|`, 71
- 2-level approach, 154
- $A*B$, 71
- $A+\{B\}$, 72
- $A+B$, 71
- Abbreviations, 208
- Abort, 123
- About, 109
- Absolute names, 57
- abstract, 174
- abstractions, 29
- absurd, 70, 135
- absurd_set, 73
- Acc, 74
- Acc_inv, 74
- Acc_rec, 74
- Add Abstract Ring, 282
- Add Abstract Semi Ring, 282
- Add Field, 159
- Add LoadPath, 116
- Add ML Path, 116
- Add Morphism, 288
- Add Printing If *ident*, 48
- Add Printing Let *ident*, 48
- Add Rec LoadPath, 116
- Add Rec ML Path, 116
- Add Ring, 158, 282
- Add Semi Ring, 158, 282
- Add Setoid, 288
- Admitted, 42, 122
- all, 69
- and, 68
- and_rec, 73
- app, 79
- applications, 29
- apply, 131
- apply ... with, 131
- Arguments Scope, 206
- Arithmetical notations, 76
- Arity, 90
- assert, 133
- Associativity, 198
- assumption, 128
- auto, 154
- autorewrite, 160
- Axiom, 32
- Axiom (and coercions), 256
- Back, 117
- Bad Magic Number, 115
- Begin Silent, 118
- β -reduction, 85, 86
- Bind Scope, 206
- binders, 29
- Binding list, 134
- BNF metasyntax, 25
- bool, 70
- bool_choice, 72
- byte-code, 213
- Calculus of (Co)Inductive Constructions, 81
- Canonical Structure, 63
- case, 143
- case ... with, 143

-
- Cases, 243
 - Cast, 30
 - cbv, 135
 - Cd, 115
 - change, 134
 - change ... in, 134
 - Check, 109
 - Choice, 72
 - Choice2, 72
 - CIC, 81
 - Clauses, 135
 - clear, 128
 - clearbody, 128
 - Close Scope, 205
 - Coercion, 64, 255
 - Coercion Local, 255, 256
 - Coercions, 64
 - and records, 257
 - and sections, 258
 - classes, 253
 - Funclass, 254
 - identity, 254
 - inheritance graph, 254
 - presentation, 253
 - Sortclass, 254
 - CoFixpoint, 40
 - CoFixpoint ... where ..., 201
 - CoInductive, 37
 - CoInductive (and coercions), 256
 - Comments, 25
 - compare, 148
 - Compiled files, 114
 - compute, 135, 136
 - congruence, 157
 - conj, 68
 - Conjecture, 32
 - Connectives, 68
 - Constant, 33
 - constructor, 139
 - constructor ... with, 139
 - Context, 83
 - context
 - in expression, 173
 - in pattern, 172
 - contradiction, 135
 - Contributions, 79
 - Conversion rules, 85
 - Conversion tactics, 135
 - coqc, 213
 - coqdep, 218
 - coqdoc, 218
 - coqide, 231
 - coq_Makefile, 218
 - coqmktop, 217
 - coq-tex, 228
 - coqtop, 213
 - cut, 133
 - cutrewrite, 146
 - Datatypes, 70
 - Debugger, 217
 - decide equality, 148
 - Declarations, 31
 - Declare Left Step, 147
 - Declare ML Module, 115
 - Declare Right Step, 147
 - decompose, 144
 - decompose record, 145
 - decompose sum, 145
 - Defined, 42, 122
 - Definition, 33, 123
 - Definitions, 33
 - Delimit Scope, 205
 - δ -reduction, 33, 85, 86
 - Dependencies, 218
 - dependent inversion, 152
 - dependent inversion ... as , 152
 - dependent inversion ... as ...
 - with, 153
 - dependent inversion ... with, 152
 - dependent inversion_clear, 152
 - dependent inversion_clear ...
 - as, 152
 - dependent inversion_clear ...
 - as ... with, 153
 - dependent inversion_clear ...
 - with, 153
 - dependent rewrite \rightarrow , 150
 - dependent rewrite \leftarrow , 151
 - Derive Dependent Inversion, 153
 - Derive Dependent
 - Inversion_clear, 153
 - Derive Inversion, 153
 - Derive Inversion_clear, 153
 - Derive Inversion_clear ... with, 153
 - destruct, 142
 - discriminate, 148, 149
 - discrR, 78
 - do, 169

- double induction, 144
- Drop, 118
- eapply, 131, 175
- eauto, 155
- elim ... using, 141
- elim ... with, 141
- Elimination
 - Empty elimination, 94
 - Singleton elimination, 94
- Elimination sorts, 93
- elimtype, 142
- Emacs, 229
- End, 49, 51, 52
- End Silent, 118
- Environment, 33, 84
- Environment variables, 214
- eq, 69
- eq_add_S, 73
- eq_ind_r, 70
- eq_rec, 73
- eq_rec_r, 70
- eq_rect, 70
- eq_rect_r, 70
- eq_S, 73
- Equality, 69
- error, 72
- η -conversion, 86
- η -reduction, 86
- Eval, 110
- eval
 - in Ltac, 173
- ex, 69
- ex2, 69
- ex_intro, 69
- ex_intro2, 69
- exact, 127
- Exc, 72
- Except, 73
- exist, 71
- exist2, 71
- exists, 72
- exists, 69, 139
- exists2, 72
- exists2, 69
- Explicitation of implicit arguments, 62
- Export, 56
- Extract Constant, 269
- Extract Inductive, 270
- Extraction, 267
 - Extraction, 110, 267
 - Extraction Inline, 269
 - Extraction Language, 268
 - Extraction Module, 267
 - Extraction NoInline, 269
- f_equal, 70
- f_equal i , 70
- Fact, 41, 123
- fail, 171
- False, 68
- false, 70
- False_rec, 73
- field, 158
- first, 170
- firstorder, 156
- firstorder using, 157
- firstorder with, 156
- firstorder tactic, 156
- Fix, 96
- fix $ident_i\{\dots\}$, 30
- fix_eq, 75
- Fix_F, 75
- Fix_F_eq, 75
- Fix_F_inv, 75
- Fixpoint, 38
- Fixpoint ... where ..., 201
- flat_map, 79
- Focus, 125
- fold, 137
- fold_left, 79
- fold_right, 79
- form, 29
- fourier, 159
- fresh
 - in Ltac, 173
- fst, 71
- fun
 - in Ltac, 172
- functional induction, 145, 178
- Functional Scheme, 165, 178
- Gallina, 25, 43
- gallina, 229
- ge, 74
- generalize, 133
- generalize dependent, 134
- Goal, 42, 121
- goal, 127
- gt, 74

-
- head, 79
 - Head normal form, 86
 - Hint, 161
 - Hint Constructors, 162
 - Hint Extern, 162
 - Hint Immediate, 161
 - Hint Resolve, 161
 - Hint Rewrite, 160
 - Hint Unfold, 162
 - Hints databases, 161
 - hnf, 136
 - Hypotheses, 32
 - Hypothesis, 32
 - Hypothesis (and coercions), 256

 - I, 68
 - ident*, 25
 - identity, 70
 - Identity Coercion, 256
 - idtac, 171
 - if ... then ... else, 46
 - IF_then_else, 69
 - iff, 69
 - Implicit Arguments, 59
 - Implicit arguments, 58
 - Import, 55
 - induction, 140
 - Inductive, 34
 - Inductive (and coercions), 256
 - Inductive definitions, 34
 - Inductive ... where ..., 201
 - Infix, 201
 - info, 174
 - injection, 149, 150
 - inl, 71
 - inleft, 72
 - inr, 71
 - inright, 72
 - Inspect, 109
 - integer*, 26
 - Interpretation scopes, 204
 - intro, 129
 - intro ... after, 130
 - intro after, 130
 - intros, 129
 - intros *intro_pattern*, 143
 - intros until, 130
 - intuition, 156
 - inversion, 151, 180
 - inversion ... as, 151
 - inversion ... as ... in, 152
 - inversion ... in, 152
 - inversion ... using, 153
 - inversion ... using ... in, 153
 - inversion_clear, 151
 - inversion_clear ... as ... in, 152
 - inversion_clear ... in, 152
 - inversion_cleardots as, 152
 - ι -reduction, 85, 86, 95, 98
 - IsSucc, 73

 - λ -calculus, 83
 - lapply, 131
 - L^AT_EX, 228
 - lazy, 135
 - le, 74
 - le_n, 74
 - le_S, 74
 - left, 72, 139
 - Lemma, 41, 122
 - length, 79
 - Let, 33
 - let
 - in Ltac, 171
 - let ... in, 46
 - let rec
 - in Ltac, 171
 - let-in, 30
 - Lexical conventions, 25
 - Libraries, 56
 - Load, 114
 - Load Verbose, 114
 - Loadpath, 115
 - Local, 123
 - local context, 121
 - Local definitions, 30
 - Locate, 113, 202
 - Locate File, 117
 - Locate Library, 117
 - Logical paths, 56
 - lt, 74
 - Ltac
 - eval, 173
 - fresh, 173
 - fun, 172
 - let, 171
 - let rec, 171
 - match, 172
 - match goal, 172

- match reverse goal, 172
- type of, 173
- Ltac, 174
- Makefile, 218
- Man pages, 229
- map, 79
- match
 - in Ltac, 172
- match...with...end, 30, 45, 92
- match goal
 - in Ltac, 172
- match reverse goal
 - in Ltac, 172
- ML-like patterns, 45, 243
- mod, 76
- Module, 50, 51
- Module Type, 51
- Modules, 50
- move, 129
- mult, 73
- mult_n_O, 73
- mult_n_Sm, 73
- Mutual Inductive, 36
- n_Sn, 73
- nat, 70
- nat_case, 74
- nat_double_ind, 74
- nat_scope, 76
- native code, 213
- None, 70
- Normal form, 86
- not, 68
- not_eq_S, 73
- Notation, 197, 208
- Notations for lists, 79
- Notations for real numbers, 77
- notT, 75
- nth, 79
- num, 26
- O, 70
- O_S, 73
- omega, 158, 263
- Opaque, 110
- Open Scope, 205
- option, 70
- Options of the command line, 214
- or, 69
- or_introl, 69
- or_intror, 69
- pair, 71
- pairT, 75
- Parameter, 32
- Parameter (and coercions), 256
- Parameters, 32
- pattern, 138
- pCIC, 81
- Peano's arithmetic, 76
- plus, 73
- plus_n_O, 73
- plus_n_Sm, 73
- pose, 132
- Positivity, 90
- Precedences, 198
- pred, 73
- pred_Sn, 73
- Predicative Calculus of (Co)Inductive Construc-
 - tions, 81
- Print, 109
- Print All, 109
- Print Classes, 257
- Print Coercion Paths, 257
- Print Coercions, 257
- Print Extraction Inline, 269
- Print Grammar constr, 199
- Print Graph, 257
- Print Hint, 164
- Print HintDb, 164
- Print Implicit, 62
- Print LoadPath, 116
- Print ML Modules, 115
- Print ML Path, 116
- Print Module, 56
- Print Module Type, 56
- Print Modules, 115
- Print Section, 109
- Print Table Printing If, 48
- Print Table Printing Let, 48
- Print Term, 109
- Print XML, 226
- prod, 71
- prodT, 75
- products, 29
- Programming, 70
- progress, 170
- proj1, 68
- proj2, 68

-
- projS1, 72
 - projS2, 72
 - Prompt, 121
 - Proof, 42, 123
 - Proof editing, 121
 - Proof General, 229
 - Proof rendering, 224
 - Proof term, 121
 - Prop, 28, 82
 - Pwd, 115

 - Qed, 42, 121
 - qualid*, 62
 - Qualified identifiers, 57
 - Quantifiers, 69
 - Quit, 118
 - quote, 154, 184

 - Record, 43
 - Recursion, 74
 - Recursive arguments, 97
 - Recursive Extraction, 267
 - Recursive Extraction Module, 267
 - red, 136
 - refine, 128, 175
 - refl_equal, 69
 - refl_identity, 70
 - reflexivity, 147
 - Remark, 41, 123
 - Remove LoadPath, 116
 - Remove Printing If *ident*, 48
 - Remove Printing Let *ident*, 48
 - rename, 129
 - repeat, 170
 - replace ... with, 146
 - Require, 114
 - Require Export, 114
 - ReservedNotation, 201
 - Reset, 117
 - Reset Extraction Inline, 269
 - Reset Initial, 117
 - Resource file, 213
 - Restart, 124
 - Restore State, 117
 - Resume, 124
 - rev, 79
 - rewrite, 146
 - rewrite ->, 146
 - rewrite -> ... in, 146
 - rewrite <-, 146
 - rewrite <- ... in, 146
 - rewrite ... in, 146
 - right, 72, 139
 - ring, 158, 279, 280

 - s, 70
 - Save, 42, 122
 - Scheme, 164, 177
 - Script file, 113
 - Search, 111
 - SearchAbout, 111
 - SearchPattern, 112
 - SearchPattern ... inside ..., 112
 - SearchPattern ... outside ..., 112
 - SearchRewrite, 113
 - Section, 49
 - Sections, 49
 - Set, 28, 82
 - set, 132
 - Set Contextual Implicit, 61
 - Set Extraction AutoInline, 269
 - Set Extraction Optimize, 268
 - Set Firstorder Depth, 157
 - Set Hyps Limit, 126
 - Set Implicit Arguments, 61
 - Set Printing All, 65
 - Set Printing Coercion, 257
 - Set Printing Coercions, 257
 - Set Printing Depth, 119
 - Set Printing Implicit, 62
 - Set Printing Notations, 201
 - Set Printing Synth, 47
 - Set Printing Width, 119
 - Set Printing Wildcard, 47
 - Set Strict Implicit, 61
 - Set Undo, 124
 - setoid_replace, 287, 288
 - setoid_rewrite, 288
 - Show, 125
 - Show Conjectures, 126
 - Show Implicits, 125
 - Show Intro, 126
 - Show Intros, 126
 - Show Proof, 125
 - Show Script, 125
 - Show Tree, 125
 - Show XML Proof, 226
 - sig, 71

-
- sig2, 71
 - sigS, 72
 - sigS2, 72
 - Silent mode, 118
 - simpl, 136
 - simpl ... in, 136
 - simple destruct, 143
 - simple induction, 142
 - simple inversion, 153
 - simple inversion ... as, 153
 - simplify_eq, 150
 - snd, 71
 - solve, 171
 - Some, 70
 - sort, 27
 - Sorts, 28, 82
 - specif, 29
 - split, 139
 - split_Rabs, 78
 - split_Rmult, 79
 - stepl, 147
 - stepr, 147
 - string, 26
 - Structure, 257
 - SubClass, 256
 - subgoal, 127
 - subst, 147
 - Substitution, 83
 - sum, 71
 - sumbool, 72
 - sumor, 72
 - Suspend, 123
 - sym_eq, 70
 - sym_not_eq, 70
 - symmetry, 147
 - symmetry in, 147
 - Syntactic Definition, 209
 - tactic, 127
 - Tactic Definition, 165
 - tactic macros, 165
 - Tacticals, 168
 - tactic₁ ; tactic₂, 169
 - abstract, 174
 - do, 169
 - fail, 171
 - first, 170
 - idtac, 171
 - info, 174
 - ||, 170
 - repeat, 170
 - solve, 171
 - try, 170
 - Tactics, 127
 - tail, 79
 - tauto, 155
 - term, 27
 - Terms, 26
 - Test Printing Depth, 119
 - Test Printing If *ident*, 48
 - Test Printing Let *ident*, 48
 - Test Printing Synth, 47
 - Test Printing Width, 119
 - Test Printing Wildcard, 47
 - Theorem, 41, 122
 - Theories, 67
 - Time, 118
 - trans_eq, 70
 - transitivity, 147
 - Transparent, 110
 - trivial, 154
 - True, 68
 - true, 70
 - try, 170
 - tt, 70
 - Type, 28, 82
 - type, 27, 29
 - type of
 - in Ltac, 173
 - Type of constructor, 90
 - type_scope, 207
 - Typing rules, 84, 128
 - App, 85, 133
 - Ax, 84
 - Const, 84
 - Conv, 86, 129, 134
 - Fix, 96
 - Lam, 85, 129
 - Let, 85, 129
 - match, 95
 - Prod, 85
 - Prod (impredicative Set), 99
 - Var, 84, 128
 - Undo, 124
 - Unfocus, 125
 - unfold, 137
 - unfold ... in, 137
 - unit, 70
 - Unset Contextual Implicit, 61

Unset Extraction AutoInline, 269
Unset Extraction Optimize, 268
Unset Hyps Limit, 126
Unset Implicit Arguments, 61
Unset Printing All, 65
Unset Printing Coercion, 257
Unset Printing Coercions, 257
Unset Printing Depth, 119
Unset Printing Implicit, 62
Unset Printing Notations, 201
Unset Printing Synth, 47
Unset Printing Width, 119
Unset Printing Wildcard, 47
Unset Strict Implicit, 61
Unset Undo, 124

value, 72
Variable, 32
Variable (and coercions), 256
Variables, 32

Well founded induction, 74
Well foundedness, 74
well_founded, 74
Write State, 118

XML exportation, 224

ζ -reduction, 86

Tactics Index

`||`, 170
`;`, 169
`:[...|...|...]`, 169

`abstract`, 174
`absurd`, 135
`apply`, 131
`apply ... with`, 131
`assert`, 133
`assumption`, 128
`auto`, 154
`autorewrite`, 160

`case`, 143
`case ... with`, 143
`cbv`, 135
`change`, 134
`change ... in`, 134
`clear`, 128
`clearbody`, 128
`compare`, 148
`compute`, 135, 136
`congruence`, 157
`constructor`, 139
`constructor ... with`, 139
`contradiction`, 135
`cut`, 133
`cutrewrite`, 146

`decide equality`, 148
`decompose`, 144
`decompose record`, 145
`decompose sum`, 145
`dependent inversion`, 152
`dependent inversion ... as`, 152
`dependent inversion ... as ... with`, 153
`dependent inversion ... with`, 152
`dependent inversion_clear`, 152
`dependent inversion_clear ... as`, 152
`dependent inversion_clear ... as ... with`, 153

`dependent inversion_clear ... with`, 153
`dependent rewrite ->`, 150
`dependent rewrite <-`, 151
`destruct`, 142
`discriminate`, 148, 149
`discrR`, 78
`do`, 169
`double induction`, 144

`eapply`, 131, 175
`eauto`, 155
`elim ... using`, 141
`elim ... with`, 141
`elimtype`, 142
`exact`, 127
`exists`, 139

`fail`, 171
`field`, 158
`first`, 170
`firstorder`, 156
`firstorder using`, 157
`firstorder with`, 156
`firstorder tactic`, 156
`fold`, 137
`fourier`, 159
`functional induction`, 145, 178

`generalize`, 133
`generalize dependent`, 134

`hnf`, 136

`idtac`, 171
`induction`, 140
`info`, 174
`injection`, 149, 150
`intro`, 129
`intro ... after`, 130
`intro after`, 130
`intros`, 129
`intros intro_pattern`, 143

intros until, 130
intuition, 156
inversion, 151, 180
inversion ... as, 151
inversion ... as ... in, 152
inversion ... in, 152
inversion ... using, 153
inversion ... using ... in, 153
inversion_clear, 151
inversion_clear ... as ... in, 152
inversion_clear ... in, 152
inversion_cleardots as, 152

lapply, 131
lazy, 135
left, 139

move, 129

omega, 158, 263

pattern, 138
pose, 132
progress, 170

quote, 154, 184

red, 136
refine, 128, 175
reflexivity, 147
rename, 129
repeat, 170
replace ... with, 146
rewrite, 146
rewrite ->, 146
rewrite -> ... in, 146
rewrite <-, 146
rewrite <- ... in, 146
rewrite ... in, 146
right, 139
ring, 158, 279, 280

set, 132
setoid_replace, 287, 288
setoid_rewrite, 288
simpl, 136
simpl ... in, 136
simple destruct, 143
simple induction, 142
simple inversion, 153
simple inversion ... as, 153
simplify_eq, 150
solve, 171
split, 139
split_Rabs, 78
split_Rmult, 79
stepl, 147
stepr, 147
subst, 147
symmetry, 147
symmetry in, 147

tauto, 155
transitivity, 147
trivial, 154
try, 170

unfold, 137
unfold ... in, 137

Vernacular Commands Index

- Abort, 123
- About, 109
- Add Abstract Ring, 282
- Add Abstract Semi Ring, 282
- Add Field, 159
- Add LoadPath, 116
- Add ML Path, 116
- Add Morphism, 288
- Add Printing If *ident*, 48
- Add Printing Let *ident*, 48
- Add Rec LoadPath, 116
- Add Rec ML Path, 116
- Add Ring, 158, 282
- Add Semi Ring, 158, 282
- Add Setoid, 288
- Admitted, 42, 122
- Arguments Scope, 206
- Axiom, 32
- Axiom (and coercions), 256

- Back, 117
- Begin Silent, 118
- Bind Scope, 206

- Canonical Structure, 63
- Cd, 115
- Check, 109
- Close Scope, 205
- Coercion, 64, 255
- Coercion Local, 255, 256
- CoFixpoint, 40
- CoFixpoint ... where ..., 201
- CoInductive, 37
- CoInductive (and coercions), 256
- Conjecture, 32

- Declare Left Step, 147
- Declare ML Module, 115
- Declare Right Step, 147
- Defined, 42, 122
- Definition, 33, 123
- Delimit Scope, 205

- Derive Dependent Inversion, 153
- Derive Dependent
 Inversion_clear, 153
- Derive Inversion, 153
- Derive Inversion_clear, 153
- Drop, 118

- End, 49, 51, 52
- End Silent, 118
- Eval, 110
- Export, 56
- Extract Constant, 269
- Extract Inductive, 270
- Extraction, 110, 267
- Extraction Inline, 269
- Extraction Language, 268
- Extraction Module, 267
- Extraction NoInline, 269

- Fact, 41, 123
- Fixpoint, 38
- Fixpoint ... where ..., 201
- Focus, 125
- Functional Scheme, 165, 178

- Goal, 42, 121

- Hint, 161
- Hint Constructors, 162
- Hint Extern, 162
- Hint Immediate, 161
- Hint Resolve, 161
- Hint Rewrite, 160
- Hint Unfold, 162
- Hypotheses, 32
- Hypothesis, 32
- Hypothesis (and coercions), 256

- Identity Coercion, 256
- Implicit Arguments, 59
- Import, 55
- Inductive, 34
- Inductive (and coercions), 256

-
- Inductive ... where ..., 201
 - Infix, 201
 - Inspect, 109
 - Lemma, 41, 122
 - Let, 33
 - Load, 114
 - Load Verbose, 114
 - Local, 123
 - Locate, 113, 202
 - Locate File, 117
 - Locate Library, 117
 - Ltac, 174
 - Module, 50, 51
 - Module Type, 51
 - Mutual Inductive, 36
 - Notation, 197, 208
 - Opaque, 110
 - Open Scope, 205
 - Parameter, 32
 - Parameter (and coercions), 256
 - Parameters, 32
 - Print, 109
 - Print All, 109
 - Print Classes, 257
 - Print Coercion Paths, 257
 - Print Coercions, 257
 - Print Extraction Inline, 269
 - Print Grammar constr, 199
 - Print Graph, 257
 - Print Hint, 164
 - Print HintDb, 164
 - Print Implicit, 62
 - Print LoadPath, 116
 - Print ML Modules, 115
 - Print ML Path, 116
 - Print Module, 56
 - Print Module Type, 56
 - Print Modules, 115
 - Print Section, 109
 - Print Table Printing If, 48
 - Print Table Printing Let, 48
 - Print Term, 109
 - Print XML, 226
 - Proof, 42, 123
 - Pwd, 115
 - Qed, 42, 121
 - Quit, 118
 - Record, 43
 - Recursive Extraction, 267
 - Recursive Extraction Module, 267
 - Remark, 41, 123
 - Remove LoadPath, 116
 - Remove Printing If *ident*, 48
 - Remove Printing Let *ident*, 48
 - Require, 114
 - Require Export, 114
 - ReservedNotation, 201
 - Reset, 117
 - Reset Extraction Inline, 269
 - Reset Initial, 117
 - Restart, 124
 - Restore State, 117
 - Resume, 124
 - Save, 42, 122
 - Scheme, 164, 177
 - Search, 111
 - SearchAbout, 111
 - SearchPattern, 112
 - SearchPattern ... inside ..., 112
 - SearchPattern ... outside ..., 112
 - SearchRewrite, 113
 - Section, 49
 - Set Contextual Implicit, 61
 - Set Extraction AutoInline, 269
 - Set Extraction Optimize, 268
 - Set Firstorder Depth, 157
 - Set Hyps Limit, 126
 - Set Implicit Arguments, 61
 - Set Printing All, 65
 - Set Printing Coercion, 257
 - Set Printing Coercions, 257
 - Set Printing Depth, 119
 - Set Printing Implicit, 62
 - Set Printing Notations, 201
 - Set Printing Synth, 47
 - Set Printing Width, 119
 - Set Printing Wildcard, 47
 - Set Strict Implicit, 61
 - Set Undo, 124
 - Show, 125
 - Show Conjectures, 126

Show Implicit, 125
Show Intro, 126
Show Intros, 126
Show Proof, 125
Show Script, 125
Show Tree, 125
Show XML Proof, 226
Structure, 257
SubClass, 256
Suspend, 123

Tactic Definition, 165
Test Printing Depth, 119
Test Printing If *ident*, 48
Test Printing Let *ident*, 48
Test Printing Synth, 47
Test Printing Width, 119
Test Printing Wildcard, 47
Theorem, 41, 122
Time, 118
Transparent, 110

Undo, 124
Unfocus, 125
Unset Contextual Implicit, 61
Unset Extraction AutoInline, 269
Unset Extraction Optimize, 268
Unset Hyps Limit, 126
Unset Implicit Arguments, 61
Unset Printing All, 65
Unset Printing Coercion, 257
Unset Printing Coercions, 257
Unset Printing Depth, 119
Unset Printing Implicit, 62
Unset Printing Notations, 201
Unset Printing Synth, 47
Unset Printing Width, 119
Unset Printing Wildcard, 47
Unset Strict Implicit, 61
Unset Undo, 124

Variable, 32
Variable (and coercions), 256
Variables, 32

Write State, 118

Index of Error Messages

- ident*₂ not found, 129
- ident*_{*i*} not found, 129
- ident* already exists, 32, 33, 42
- ident* not found, 128

- A record cannot be recursive, 45
- A Setoid Theory is already declared for *A*, 288
- All terms must have the same type, 280
- already exists, 122
- Argument of match does not evaluate to a term, 172
- Attempt to save an incomplete proof, 122

- Bad magic number, 115
- Bound head variable, 161

- Can't find file *ident* on loadpath, 114
- Can't find module *toto* on loadpath, 115
- cannot be used as a hint, 161, 162
- Cannot find the source class of *qualid*, 255
- Cannot infer a term for this placeholder, 59, 128
- Cannot load *ident*: no physical path bound to *dirpath*, 114
- Cannot move *ident*₁ after *ident*₂: it depends on *ident*₂, 129
- Cannot move *ident*₁ after *ident*₂: it occurs in *ident*₂, 129
- Cannot recognize *class*₁ as a source class of *qualid*, 255
- Cannot refine to conclusions with meta-variables, 140
- Cannot solve the goal, 171

- Delta must be specified before, 136
- does not denote an evaluable constant, 137
- does not respect the inheritance uniform condition, 255
- Don't know what to do with this goal, 280

- Failed to progress, 170
- File not found on loadpath : *string*, 115
- Found target class *class* instead of *class*₂, 255
- Funclass cannot be a source class, 255

- generated subgoal *term*' has metavariables in it, 131
- goal does not satisfy the expected preconditions, 150

- I couldn't solve goal, 157
- I don't know how to handle dependent equality, 157
- Impossible to unify ... with ..., 147
- Impossible to unify ... with ..., 131, 142
- In environment ... the term: *term*₂ does not have type *term*₁, 33
- invalid argument, 128
- is already a coercion, 255
- is already used, 129
- is not a function, 255
- is not a module, 56
- is not a projectable equality, 150
- is not an inductive type, 162
- is used in the conclusion, 128
- is used in the hypothesis, 128

- Loading of ML object file forbidden in a native Coq, 115

- Module/section *module* not found, 111
- must be a transparent constant, 256

- name *ident* is already used, 130
- No applicable tactic, 170
- No Declared Ring Theory for *term*., 280
- No discriminable equalities, 149
- No focused proof, 121, 125
- No focused proof (No proof-editing in progress), 123, 124
- No focused proof to restart, 124
- No matching clauses for match, 172
- No matching clauses for match goal, 173
- No product even after head-reduction, 129, 130
- No proof-editing in progress, 124
- No such assumption, 128, 135
- no such entry, 117

- No such goal, 125
- No such hypothesis, 130, 138, 146
- No such hypothesis in current goal, 130
- No such label *ident*, 51
- No such proof, 124
- Non exhaustive pattern-matching, 251
- Non strictly positive occurrence of *ident* in *type*, 35
- not a context variable, 173
- not a defined object, 109
- Not a discriminable equality, 148
- Not a proposition or a type, 133
- Not a valid (semi)ring theory, 282
- Not a valid setoid theory, 288
- Not an equation, 150
- Not an exact proof, 128
- Not an inductive product, 139, 140
- Not convertible, 134
- not declared, 162, 255
- Not enough constructors, 139
- Not reducible, 136
- Not the right number of dependent arguments, 141
- Not the right number of missing arguments, 131
- Nothing to rewrite in *ident*, 146

- omega can't solve this system, 264
- omega: Can't solve a goal with equality on *type*, 264
- omega: Can't solve a goal with non-linear products, 264
- omega: Can't solve a goal with proposition variables, 264
- omega: Not a quantifier-free goal, 264
- omega: Unrecognized atomic proposition: *prop*, 264
- omega: Unrecognized predicate or connective: *ident*, 264
- omega: Unrecognized proposition, 264

- Proof is not complete, 174

- quote: not a simple fixpoint, 154, 185

- Reached begin of command history, 117

- Signature components for label *ident* do not match, 51
- Sortclass cannot be a source class, 255

- Tactic Failure message (level *n*), 171
- Tactic generated a subgoal identical to the original goal, 146
- The conclusion is not a substitutive equation, 147
- The conclusion of *type* is not valid; it must be built from *ident*, 35
- The reference *qualid* was not found in the current environment, 110, 111
- the term *form* has type ... which should be Set, Prop or Type, 121, 122
- The term *term* is already declared as a morphism, 288
- The term *term* is not a product, 288
- The term *term* should not be a dependent product, 288
- The term provided does not end with an equation, 146
- The *numth* argument of *ident* must be *ident*' in *type*, 36
- This goal is not an equality, 281
- This is not the last opened module, 51
- This is not the last opened module type, 52
- This is not the last opened section, 50

- Undo stack would be exhausted, 124

List of Figures

1.1	Syntax of terms	27
1.2	Syntax of terms (continued)	28
1.3	Syntax of sentences	31
2.1	Syntax for the definition of <code>Record</code>	43
2.2	Syntax of <code>Record</code> projections	45
2.3	Syntax of modules	50
2.4	Syntax for explicitations of implicit arguments	62
3.1	Notations in the initial state	68
3.2	Syntax of formulas	68
3.3	Syntax of datatypes and specifications	71
3.4	Definition of the scope for integer arithmetics (<code>Z_ scope</code>)	77
3.5	Definition of the scope for natural numbers (<code>nat_ scope</code>)	77
3.6	Definition of the scope for real arithmetics (<code>R_ scope</code>)	78
3.7	Definition of the scope for lists (<code>list_ scope</code>)	79
9.1	Syntax of the tactic language	168
9.2	Syntax of the tactic language (continued)	169
9.3	Tactic toplevel definitions	169
10.1	Definition of the permutation predicate	188
10.2	Permutation tactic	189
10.3	Deciding intuitionistic propositions (1)	189
10.4	Deciding intuitionistic propositions (2)	190
10.5	Type isomorphism axioms	191
10.6	Type isomorphism tactic (1)	192
10.7	Type isomorphism tactic (2)	193
11.1	Syntax of the variants of <code>Notation</code>	202
14.1	COQIDE main screen	232
14.2	COQIDE: the query window	233
16.1	Syntax of classes	254