The Coq Reference Manual

Release 8.11.2

The Coq Development Team

May 27, 2020

CONTENTS

CHAPTER

ONE

INTRODUCTION

This document is the Reference Manual of the Coq proof assistant. To start using Coq, it is advised to first read a tutorial. Links to several tutorials can be found at https://coq.inria.fr/documentation and https://github.com/coq/coq/wiki#coq-tutorials

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 specifications. 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.

- In interactive mode, users can develop their theories and proofs step by step, and query the system for available theorems and definitions. The interactive mode is generally run with the help of an IDE, such as CoqIDE, documented in *Coq Integrated Development Environment*, Emacs with Proof-General *[Asp00]*[4], or jsCoq to run Coq in your browser (see https://github.com/ejgallego/jscoq). The `coqtop` read-eval-print-loop can also be used directly, for debugging purposes.
- The compiled mode acts as a proof checker taking a file containing a whole development in order to ensure its correctness. Moreover, Coq's compiler provides an output file containing a compact representation of its input. The compiled mode is run with the `coqc` command.

See also:

The Coq commands.

1.1 How to read this book

This is a Reference Manual, so it is not intended for continuous reading. We recommend using the various indexes to quickly locate the documentation you are looking for. There is a global index, and a number of specific indexes for tactics, vernacular commands, and error messages and warnings. Nonetheless, the manual has some structure that is explained below.

[4] Proof-General is available at https://proofgeneral.github.io/. Optionally, you can enhance it with the minor mode Company-Coq *[PCC16]* (see https://github.com/cpitclaudel/company-coq).

- The first part describes the specification language, Gallina. Chapters *The Gallina specification language* and *Extensions of Gallina* describe the concrete syntax as well as the meaning of programs, theorems and proofs in the Calculus of Inductive Constructions. Chapter *The Coq library* describes the standard library of Coq. Chapter *Calculus of Inductive Constructions* is a mathematical description of the formalism. Chapter *The Module System* describes the module system.
- The second part describes the proof engine. It is divided into several chapters. Chapter *Vernacular commands* 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 *Proof handling*. In Chapter *Tactics*, all commands that realize one or more steps of the proof are presented: we call them *tactics*. The legacy language to combine these tactics into complex proof strategies is given in Chapter *Ltac*. The currently experimental language that will eventually replace Ltac is presented in Chapter *Ltac2*. Examples of tactics are described in Chapter *Detailed examples of tactics*. Finally, the SSReflect proof language is presented in Chapter *The SSReflect proof language*.
- The third part describes how to extend the syntax of Coq in Chapter *Syntax extensions and interpretation scopes* and how to define new induction principles in Chapter *Proof schemes*.
- In the fourth part more practical tools are documented. First in Chapter *The Coq commands*, the usage of `coqc` (batch mode) and `coqtop` (interactive mode) with their options is described. Then, in Chapter *Utilities*, various utilities that come with the Coq distribution are presented. Finally, Chapter *Coq Integrated Development Environment* describes CoqIDE.
- The fifth part documents a number of advanced features, including coercions, canonical structures, typeclasses, program extraction, and specialized solvers and tactics. See the table of contents for a complete list.

1.2 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:

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 available at https://coq.inria.fr/stdlib/.

1.3 License

This material (the Coq Reference Manual) may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at http://www.opencontent.org/openpub). Options A and B are not elected.

CHAPTER

TWO

EARLY HISTORY OF COQ

2.1 Historical roots

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[5] 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 language, 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 semidecision 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

[5] At the time of writing, i.e. 1995.

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.

2.2 Versions 1 to 5

Note: This summary was written in 1995 together with the previous section and formed the initial version of the Credits chapter.

A more comprehensive description of these early versions is available in the following subsections, which come from a document written in September 2015 by Gérard Huet, Thierry Coquand and Christine Paulin.

A first implementation of CoC was started in 1984 by G. Huet and T. Coquand. Its implementation language was CAML, a functional programming language from the ML family designed at INRIA in Rocquencourt. The core of this system was a proof-checker for CoC seen as a typed λ-calculus, called the *Constructive Engine*. This engine was operated through a high-level notation permitting the declaration of axioms and parameters, the definition of mathematical types and objects, and the explicit construction of proof objects encoded as λ-terms. A section mechanism, designed and implemented by G. Dowek, allowed hierarchical developments of mathematical theories. This high-level language was called the *Mathematical Vernacular*. Furthermore, an interactive *Theorem Prover* permitted the incremental construction of proof trees in a top-down manner, subgoaling recursively and backtracking from dead-ends. The theorem prover executed tactics written in CAML, in the LCF fashion. A basic set of tactics was predefined, which the user could extend by his own specific tactics. This system (Version 4.10) was released in 1989. Then, the system was extended to deal with the new calculus with inductive types by C. Paulin, with corresponding new tactics for proofs by induction. A new standard set of tactics was streamlined, and the vernacular extended for tactics execution.

A package to compile programs extracted from proofs to actual computer programs in CAML or some other functional language was designed and implemented by B. Werner. A new user-interface, relying on a CAML-X interface by D. de Rauglaudre, was designed and implemented by A. Felty. It allowed operation of the theorem-prover through the manipulation of windows, menus, mouse-sensitive buttons, and other widgets. This system (Version 5.6) was released in 1991.

Coq was ported to the new implementation Caml-light of X. Leroy and D. Doligez by D. de Rauglaudre (Version 5.7) in 1992. A new version of Coq was then coordinated by C. Murthy, with new tools designed by C. Parent to prove properties of ML programs (this methodology is dual to program extraction) and a new user-interaction loop. This system (Version 5.8) was released in May 1993. A Centaur interface CTCoq was then developed by Y. Bertot from the Croap project from INRIA-Sophia-Antipolis.

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

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

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

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

Rocquencourt, Feb. 1st 1995
Gérard Huet

2.2.1 Version 1

This software is a prototype type-checker for a higher-order logical formalism known as the Theory of Constructions, presented in his PhD thesis by Thierry Coquand, with influences from Girard’s system F and de Bruijn’s Automath. The metamathematical analysis of the system is the PhD work of Thierry Coquand. The software is mostly the work of Gérard Huet. Most of the mathematical examples verified with the software are due to Thierry Coquand.

The programming language of the CONSTR software (as it was called at the time) was a version of ML adapted from the Edinburgh LCF system and running on a LISP backend. The main improvements from the original LCF ML were that ML was compiled rather than interpreted (Gérard Huet building on the original translator by Lockwood Morris), and that it was enriched by recursively defined types (work of Guy Cousineau). This ancestor of CAML was used and improved by Larry Paulson for his implementation of Cambridge LCF.

Software developments of this prototype occurred from late 1983 to early 1985.

Version 1.10 was frozen on December 22nd 1984. It is the version used for the examples in Thierry Coquand's thesis, defended on January 31st 1985. There was a unique binding operator, used both for universal quantification (dependent product) at the level of types and functional abstraction (λ) at the level of terms/proofs, in the manner of Automath. Substitution (λ-reduction) was implemented using de Bruijn's indexes.

Version 1.11 was frozen on February 19th, 1985. It is the version used for the examples in the paper: T. Coquand, G. Huet. *Constructions: A Higher Order Proof System for Mechanizing Mathematics* [CH85].

Christine Paulin joined the team at this point, for her DEA research internship. In her DEA memoir (August 1985) she presents developments for the *lambo* function $-$ $\text{lambo}(f)(n)$ computes the minimal m such that $f(m)$ is greater than n, for f an increasing integer function, a challenge for constructive mathematics. She also encoded the majority voting algorithm of Boyer and Moore.

2.2.2 Version 2

The formal system, now renamed as the *Calculus of Constructions*, was presented with a proof of consistency and comparisons with proof systems of Per Martin Löf, Girard, and the Automath family of N. de Bruijn, in the paper: T. Coquand and G. Huet. *The Calculus of Constructions* [CH86b].

An abstraction of the software design, in the form of an abstract machine for proof checking, and a fuller sequence of mathematical developments was presented in: T. Coquand, G. Huet. *Concepts Mathématiques et Informatiques Formalisés dans le Calcul des Constructions* [CH86a].

Version 2.8 was frozen on December 16th, 1985, and served for developing the examples in the above papers.

This calculus was then enriched in version 2.9 with a cumulative hierarchy of universes. Universe levels were initially explicit natural numbers. Another improvement was the possibility of automatic synthesis of implicit type arguments, relieving the user of tedious redundant declarations.

Christine Paulin wrote an article *Algorithm development in the Calculus of Constructions* [Moh86]. Besides *lambo* and *majority*, she presents *quicksort* and a text formatting algorithm.

Version 2.13 of the Calculus of Constructions with universes was frozen on June 25th, 1986.

A synthetic presentation of type theory along constructive lines with ML algorithms was given by Gérard Huet in his May 1986 CMU course notes *Formal Structures for Computation and Deduction*. Its chapter *Induction and Recursion in the Theory of Constructions* was presented as an invited paper at the Joint Conference on Theory and Practice of Software Development TAPSOFT'87 at Pise in March 1987, and published as *Induction Principles Formalized in the Calculus of Constructions* [Hue88].

2.2.3 Version 3

This version saw the beginning of proof automation, with a search algorithm inspired from PROLOG and the applicative logic programming programs of the course notes *Formal structures for computation and deduction*. The search algorithm was implemented in ML by Thierry Coquand. The proof system could thus be used in two modes: proof verification and proof synthesis, with tactics such as `AUTO`.

The implementation language was now called CAML, for Categorical Abstract Machine Language. It used as backend the LLM3 virtual machine of Le Lisp by Jérôme Chailloux. The main developers of CAML were Michel Mauny, Ascander Suarez and Pierre Weis.

V3.1 was started in the summer of 1986, V3.2 was frozen at the end of November 1986. V3.4 was developed in the first half of 1987.

Thierry Coquand held a post-doctoral position in Cambridge University in 1986-87, where he developed a variant implementation in SML, with which he wrote some developments on fixpoints in Scott's domains.

2.2.4 Version 4

This version saw the beginning of program extraction from proofs, with two varieties of the type `Prop` of propositions, indicating constructive intent. The proof extraction algorithms were implemented by Christine Paulin-Mohring.

V4.1 was frozen on July 24th, 1987. It had a first identified library of mathematical developments (directory `exemples`), with libraries `Logic` (containing impredicative encodings of intuitionistic logic and algebraic primitives for booleans, natural numbers and list), `Peano` developing second-order Peano arithmetic, `Arith` defining addition, multiplication, euclidean division and factorial. Typical developments were the Knaster-Tarski theorem and Newman's lemma from rewriting theory.

V4.2 was a joint development of a team consisting of Thierry Coquand, Gérard Huet and Christine Paulin-Mohring. A file V4.2.log records the log of changes. It was frozen on September 1987 as the last version implemented in CAML 2.3, and V4.3 followed on CAML 2.5, a more stable development system.

V4.3 saw the first top-level of the system. Instead of evaluating explicit quotations, the user could develop his mathematics in a high-level language called the mathematical vernacular (following Automath terminology). The user could develop files in the vernacular notation (with `.v` extension) which were now separate from the `ml` sources of the implementation. Gilles Dowek joined the team to develop the vernacular language as his DEA internship research.

A notion of sticky constant was introduced, in order to keep names of lemmas when local hypotheses of proofs were discharged. This gave a notion of global mathematical environment with local sections.

Another significant practical change was that the system, originally developed on the VAX central computer of our lab, was transferred on SUN personal workstations, allowing a level of distributed development. The extraction algorithm was modified, with three annotations `Pos`, `Null` and `Typ` decorating the sorts `Prop` and `Type`.

Version 4.3 was frozen at the end of November 1987, and was distributed to an early community of users (among those were Hugo Herbelin and Loic Colson).

V4.4 saw the first version of (encoded) inductive types. Now natural numbers could be defined as:

```
[source, coq]
Inductive NAT : Prop = O : NAT | Succ : NAT->NAT.
```

These inductive types were encoded impredicatively in the calculus, using a subsystem *rec* due to Christine Paulin. V4.4 was frozen on March 6th 1988.

Version 4.5 was the first one to support inductive types and program extraction. Its banner was *Calcul des Constructions avec Réalisations et Synthèse*. The vernacular language was enriched to accommodate extraction commands.

The verification engine design was presented as: G. Huet. *The Constructive Engine*. Version 4.5. Invited Conference, 2nd European Symposium on Programming, Nancy, March 88. The final paper, describing the V4.9 implementation, appeared in: A perspective in Theoretical Computer Science, Commemorative Volume in memory of Gift Siromoney, Ed. R. Narasimhan, World Scientific Publishing, 1989.

Version 4.5 was demonstrated in June 1988 at the YoP Institute on Logical Foundations of Functional Programming organized by Gérard Huet at Austin, Texas.

Version 4.6 was started during the summer of 1988. Its main improvement was the complete rehaul of the proof synthesis engine by Thierry Coquand, with a tree structure of goals.

Its source code was communicated to Randy Pollack on September 2nd 1988. It evolved progressively into LEGO, proof system for Luo's formalism of Extended Calculus of Constructions.

The discharge tactic was modified by Gérard Huet to allow for inter-dependencies in discharged lemmas. Christine Paulin improved the inductive definition scheme in order to accommodate predicates of any arity.

Version 4.7 was started on September 6th, 1988.

This version starts exploiting the CAML notion of module in order to improve the modularity of the implementation. Now the term verifier is identified as a proper module Machine, which the structure of its internal data structures being hidden and thus accessible only through the legitimate operations. This machine (the constructive engine) was the trusted core of the implementation. The proof synthesis mechanism was a separate proof term generator. Once a complete proof term was synthesized with the help of tactics, it was entirely re-checked by the engine. Thus there was no need to certify the tactics, and the system took advantage of this fact by having tactics ignore the universe levels, universe consistency check being relegated to the final type-checking pass. This induced a certain puzzlement in early users who saw, after a successful proof search, their `QED` followed by silence, followed by a failure message due to a universe inconsistency…

The set of examples comprise set theory experiments by Hugo Herbelin, and notably the Schroeder-Bernstein theorem.

Version 4.8, started on October 8th, 1988, saw a major re-implementation of the abstract syntax type `constr`, separating variables of the formalism and metavariables denoting incomplete terms managed by the search mechanism. A notion of level (with three values `TYPE`, `OBJECT` and `PROOF`) is made explicit and a type judgement clarifies the constructions, whose implementation is now fully explicit. Structural equality is speeded up by using pointer equality, yielding spectacular improvements. Thierry Coquand adapts the proof synthesis to the new representation, and simplifies pattern matching to first-order predicate calculus matching, with important performance gain.

A new representation of the universe hierarchy is then defined by Gérard Huet. Universe levels are now implemented implicitly, through a hidden graph of abstract levels constrained with an order relation. Checking acyclicity of the graph insures well-foundedness of the ordering, and thus consistency. This was documented in a memo *Adding Type:Type to the Calculus of Constructions* which was never published.

The development version is released as a stable 4.8 at the end of 1988.

Version 4.9 is released on March 1st 1989, with the new ”elastic” universe hierarchy.

The spring of 1989 saw the first attempt at documenting the system usage, with a number of papers describing the formalism:

- *Metamathematical Investigations of a Calculus of Constructions*, by Thierry Coquand [Coq89],
- *Inductive definitions in the Calculus of Constructions*, by Christine Paulin-Mohrin,
- *Extracting Fω's programs from proofs in the Calculus of Constructions*, by Christine Paulin-Mohring* [PM89],
- *The Constructive Engine*, by Gérard Huet [Hue89],

as well as a number of user guides:

- *A short user's guide for the Constructions*, Version 4.10, by Gérard Huet
- *A Vernacular Syllabus*, by Gilles Dowek.
- *The Tactics Theorem Prover, User's guide*, Version 4.10, by Thierry Coquand.

Stable V4.10, released on May 1st, 1989, was then a mature system, distributed with CAML V2.6.

In the mean time, Thierry Coquand and Christine Paulin-Mohring had been investigating how to add native inductive types to the Calculus of Constructions, in the manner of Per Martin-Löf's Intuitionistic Type Theory. The impredicative encoding had already been presented in: F. Pfenning and C. Paulin-Mohring. *Inductively defined types in the Calculus of Constructions [PPM89]*. An extension of the calculus with primitive inductive types appeared in: T. Coquand and C. Paulin-Mohring. *Inductively defined types [CP90]*.

This led to the Calculus of Inductive Constructions, logical formalism implemented in Versions 5 upward of the system, and documented in: C. Paulin-Mohring. *Inductive Definitions in the System Coq - Rules and Properties [PM93b]*.

The last version of CONSTR is Version 4.11, which was last distributed in the spring of 1990. It was demonstrated at the first workshop of the European Basic Research Action Logical Frameworks In Sophia Antipolis in May 1990.

2.2.5 Version 5

At the end of 1989, Version 5.1 was started, and renamed as the system Coq for the Calculus of Inductive Constructions. It was then ported to the new stand-alone implementation of ML called Caml-light.

In 1990 many changes occurred. Thierry Coquand left for Chalmers University in Göteborg. Christine Paulin-Mohring took a CNRS researcher position at the LIP laboratory of École Normale Supérieure de Lyon. Project Formel was terminated, and gave rise to two teams: Cristal at INRIA-Roquencourt, that continued developments in functional programming with Caml-light then OCaml, and Coq, continuing the type theory research, with a joint team headed by Gérard Huet at INRIA-Rocquencourt and Christine Paulin-Mohring at the LIP laboratory of CNRS-ENS Lyon.

Chetan Murthy joined the team in 1991 and became the main software architect of Version 5. He completely rehauled the implementation for efficiency. Versions 5.6 and 5.8 were major distributed versions, with complete documentation and a library of users' developments. The use of the RCS revision control system, and systematic ChangeLog files, allow a more precise tracking of the software developments.

September 2015 +
Thierry Coquand, Gérard Huet and Christine Paulin-Mohring.

2.3 Versions 6

2.3.1 Version 6.1

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

The new version provides powerful tools for easier developments.

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

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

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

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

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

Lyon, Nov. 18th 1996
Christine Paulin

2.3.2 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 reduction 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 the 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 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

2.3.3 Version 6.3

The main changes in version V6.3 were 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 fixed 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

2.4 Versions 7

2.4.1 Summary of changes

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 efficiency of the reduction algorithm 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, reflection-based version of the Omega decision procedure.

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

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

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

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

The development was coordinated by C. Paulin.

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

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

Orsay, May. 2002
Hugo Herbelin & Christine Paulin

2.4.2 Details of changes in 7.0 and 7.1

Notes:

- items followed by (**) are important sources of incompatibilities
- items followed by (*) may exceptionally be sources of incompatibilities
- items followed by (+) have been introduced in version 7.0

Main novelties

References are to Coq 7.1 reference manual

- New primitive let-in construct (see sections 1.2.8 and)
- Long names (see sections 2.6 and 2.7)

- New high-level tactic language (see chapter 10)
- Improved search facilities (see section 5.2)
- New extraction algorithm managing the Type level (see chapter 17)
- New rewriting tactic for arbitrary equalities (see chapter 19)
- New tactic Field to decide equalities on commutative fields (see 7.11)
- New tactic Fourier to solve linear inequalities on reals numbers (see 7.11)
- New tactics for induction/case analysis in ”natural” style (see 7.7)
- Deep restructuration of the code (safer, simpler and more efficient)
- Export of theories to XML for publishing and rendering purposes (see http://www.cs.unibo.it/helm)

Details of changes

Language: new ”let-in” construction

- New construction for local definitions (let-in) with syntax [x:=u]t (*)(+)
- Local definitions allowed in Record (a.k.a. record à la Randy Pollack)

Language: long names

- Each construction has a unique absolute names built from a base name, the name of the module in which they are defined (Top if in coqtop), and possibly an arbitrary long sequence of directory (e.g. ”Coq.Lists.PolyList.flat_map” where ”Coq” means that ”flat_map” is part of Coq standard library, ”Lists” means it is defined in the Lists library and ”PolyList” means it is in the file Polylist) (+)
- Constructions can be referred by their base name, or, in case of conflict, by a ”qualified” name, where the base name is prefixed by the module name (and possibly by a directory name, and so on). A fully qualified name is an absolute name which always refer to the construction it denotes (to preserve the visibility of all constructions, no conflict is allowed for an absolute name) (+)
- Long names are available for modules with the possibility of using the directory name as a component of the module full name (with option -R to coqtop and coqc, or command Add LoadPath) (+)
- Improved conflict resolution strategy (the Unix PATH model), allowing more constructions to be referred just by their base name

Language: miscellaneous

- The names of variables for Record projections _and_ for induction principles (e.g. sum_ind) is now based on the first letter of their type (main source of incompatibility) (**)(+)
- Most typing errors have now a precise location in the source (+)
- Slightly different mechanism to solve ”?” (*)(+)
- More arguments may be considered implicit at section closing (*)(+)
- Bug with identifiers ended by a number greater than 2^30 fixed (+)

- New visibility discipline for Remark, Fact and Local: Remark's and Fact's now survive at the end of section, but are only accessible using a qualified names as soon as their strength expires; Local's disappear and are moved into local definitions for each construction persistent at section closing

Language: Cases

- Cases no longer considers aliases inferable from dependencies in types (*)(+)
- A redundant clause in Cases is now an error (*)

Reduction

- New reduction flags "Zeta" and "Evar" in Eval Compute, for inlining of local definitions and instantiation of existential variables
- Delta reduction flag does not perform Zeta and Evar reduction any more (*)
- Constants declared as opaque (using Qed) can no longer become transparent (a constant intended to be alternatively opaque and transparent must be declared as transparent (using Defined)); a risk exists (until next Coq version) that Simpl and Hnf reduces opaque constants (*)

New tactics

- New set of tactics to deal with types equipped with specific equalities (a.k.a. Setoids, e.g. nat equipped with eq_nat) [by C. Renard]
- New tactic Assert, similar to Cut but expected to be more user-friendly
- New tactic NewDestruct and NewInduction intended to replace Elim and Induction, Case and Destruct in a more user-friendly way (see restrictions in the reference manual)
- New tactic ROmega: an experimental alternative (based on reflexion) to Omega [by P. Crégut]
- New tactic language Ltac (see reference manual) (+)
- New versions of Tauto and Intuition, fully rewritten in the new Ltac language; they run faster and produce more compact proofs; Tauto is fully compatible but, in exchange of a better uniformity, Intuition is slightly weaker (then use Tauto instead) (**)(+)
- New tactic Field to decide equalities on commutative fields (as a special case, it works on real numbers) (+)
- New tactic Fourier to solve linear inequalities on reals numbers [by L. Pottier] (+)
- New tactics dedicated to real numbers: DiscrR, SplitRmult, SplitAbsolu (+)

Changes in existing tactics

- Reduction tactics in local definitions apply only to the body
- New syntax of the form "Compute in Type of H." to require a reduction on the types of local definitions
- Inversion, Injection, Discriminate, ... apply also on the quantified premises of a goal (using the "Intros until" syntax)
- Decompose has been fixed but hypotheses may get different names (*)(+)

- Tauto now manages uniformly hypotheses and conclusions of the form `t=t` which all are considered equivalent to `True`. Especially, Tauto now solves goals of the form `H : ~ t = t |- A`.
- The ”Let” tactic has been renamed ”LetTac” and is now based on the primitive ”let-in” (+)
- Elim can no longer be used with an elimination schema different from the one defined at definition time of the inductive type. To overload an elimination schema, use ”Elim <hyp> using <name of the new schema>” (*)(+)
- Simpl no longer unfolds the recursive calls of a mutually defined fixpoint (*)(+)
- Intro now fails if the hypothesis name already exists (*)(+)
- ”Require Prolog” is no longer needed (i.e. it is available by default) (*)(+)
- Unfold now fails on a non unfoldable identifier (*)(+)
- Unfold also applies on definitions of the local context
- AutoRewrite now deals only with the main goal and it is the purpose of Hint Rewrite to deal with generated subgoals (+)
- Redundant or incompatible instantiations in Apply ... with ... are now correctly managed (+)

Efficiency

- Excessive memory uses specific to V7.0 fixed
- Sizes of .vo files vary a lot compared to V6.3 (from -30% to +300% depending on the developments)
- An improved reduction strategy for lazy evaluation
- A more economical mechanism to ensure logical consistency at the Type level; warning: this is experimental and may produce ”universes” anomalies (please report)

Concrete syntax of constructions

- Only identifiers starting with ”_” or a letter, and followed by letters, digits, ”_” or ”’” are allowed (e.g. ”$” and ”@” are no longer allowed) (*)
- A multiple binder like (a:A)(a,b:(P a))(Q a) is no longer parsed as (a:A)(a0:(P a))(b:(P a))(Q a0) but as (a:A)(a0:(P a))(b:(P a0))(Q a0) (*)(+)
- A dedicated syntax has been introduced for Reals (e.g `3+1/x`) (+)
- Pretty-printing of Infix notations fixed. (+)

Parsing and grammar extension

- More constraints when writing ast
  - ”{...}” and the macros $LIST, $VAR, etc. now expect a metavariable (an identifier starting with $) (*)
  - **identifiers should starts with a letter or ”_” and be followed** by letters, digits, ”_” or ”’” (other characters are still supported but it is not advised to use them) (*)(+)
- Entry ”command” in ”Grammar” and quotations (<<...>> stuff) is renamed ”constr” as in ”Syntax” (+)

- New syntax "[" sentence_1 ... sentence_n"]." to group sentences (useful for Time and to write grammar rules abbreviating several commands) (+)
- The default parser for actions in the grammar rules (and for patterns in the pretty-printing rules) is now the one associated to the grammar (i.e. vernac, tactic or constr); no need then for quotations as in <:vernac:<...>>; to return an "ast", the grammar must be explicitly typed with tag ": ast" or ": ast list", or if a syntax rule, by using <<...>> in the patterns (expression inside these angle brackets are parsed as "ast"); for grammars other than vernac, tactic or constr, you may explicitly type the action with tags ": constr", ": tactic", or ":vernac" (**)(+)
- Interpretation of names in Grammar rule is now based on long names, which allows to avoid problems (or sometimes tricks;) related to overloaded names (+)

New commands

- New commands "Print XML All", "Show XML Proof", ... to show or export theories to XML to be used with Helm's publishing and rendering tools (see http://www.cs.unibo.it/helm) (by Claudio Sacerdoti Coen) (+)
- New commands to manually set implicit arguments (+)
  - "Implicits ident." to activate the implicit arguments mode just for ident
  - **"Implicits ident [num1 num2 ...]." to explicitly give which** arguments have to be considered as implicit
- New SearchPattern/SearchRewrite (by Yves Bertot) (+)
- New commands "Debug on"/"Debug off" to activate/deactivate the tactic language debugger (+)
- New commands to map physical paths to logical paths (+) - Add LoadPath physical_dir as logical_dir - Add Rec LoadPath physical_dir as logical_dir

Changes in existing commands

- Generalization of the usage of qualified identifiers in tactics and commands about globals, e.g. Decompose, Eval Delta; Hints Unfold, Transparent, Require
- Require synchronous with Reset; Require's scope stops at Section ending (*)
- For a module indirectly loaded by a "Require" but not exported, the command "Import module" turns the constructions defined in the module accessible by their short name, and activates the Grammar, Syntax, Hint, ... declared in the module (+)
- The scope of the "Search" command can be restricted to some modules (+)
- Final dot in command (full stop/period) must be followed by a blank (newline, tabulation or whitespace) (+)
- Slight restriction of the syntax for Cbv Delta: if present, option [-myconst] must immediately follow the Delta keyword (*)(+)
- SearchIsos currently not supported
- Add ML Path is now implied by Add LoadPath (+)
- New names for the following commands (+)

  AddPath -> Add LoadPath Print LoadPath -> Print LoadPath DelPath -> Remove LoadPath AddRecPath -> Add Rec LoadPath Print Path -> Print Coercion Paths

Implicit Arguments On -> Set Implicit Arguments Implicit Arguments Off -> Unset Implicit Arguments

Begin Silent -> Set Silent End Silent -> Unset Silent.

Tools

- coqtop (+)
  - Two executables: coqtop.byte and coqtop.opt (if supported by the platform)
  - coqtop is a link to the more efficient executable (coqtop.opt if present)
  - option -full is obsolete (+)
- do_Makefile renamed into coq_makefile (+)
- New option -R to coqtop and coqc to map a physical directory to a logical one (+)
- coqc no longer needs to create a temporary file
- No more warning if no initialization file .coqrc exists

Extraction

- New algorithm for extraction able to deal with ”Type” (+) (by J.-C. Filliâtre and P. Letouzey)

Standard library

- New library on maps on integers (IntMap, contributed by Jean Goubault)
- New lemmas about integer numbers [ZArith]
- New lemmas and a ”natural” syntax for reals [Reals] (+)
- Exc/Error/Value renamed into Option/Some/None (*)

New user contributions

- Constructive complex analysis and the Fundamental Theorem of Algebra [FTA] (Herman Geuvers, Freek Wiedijk, Jan Zwanenburg, Randy Pollack, Henk Barendregt, Nijmegen)
- A new axiomatization of ZFC set theory [Functions_in_ZFC] (C. Simpson, Sophia-Antipolis)
- Basic notions of graph theory [GRAPHS-BASICS] (Jean Duprat, Lyon)
- A library for floating-point numbers [Float] (Laurent Théry, Sylvie Boldo, Sophia-Antipolis)
- Formalisation of CTL and TCTL temporal logic [CtlTctl] (Carlos Daniel Luna,Montevideo)
- Specification and verification of the Railroad Crossing Problem in CTL and TCTL [RailroadCrossing] (Carlos Daniel Luna,Montevideo)
- P-automaton and the ABR algorithm [PAutomata] (Christine Paulin, Emmanuel Freund, Orsay)
- Semantics of a subset of the C language [MiniC] (Eduardo Giménez, Emmanuel Ledinot, Suresnes)
- Correctness proofs of the following imperative algorithms: Bresenham line drawing algorithm [Bresenham], Marché’s minimal edition distance algorithm [Diff] (Jean-Christophe Filliâtre, Orsay)

- Correctness proofs of Buchberger's algorithm [Buchberger] and RSA cryptographic algorithm [Rsa] (Laurent Théry, Sophia-Antipolis)
- Correctness proof of Stalmarck tautology checker algorithm [Stalmarck] (Laurent Théry, Pierre Letouzey, Sophia-Antipolis)

2.4.3 Details of changes in 7.2

Language

- Automatic insertion of patterns for local definitions in the type of the constructors of an inductive types (for compatibility with V6.3 let-in style)
- Coercions allowed in Cases patterns
- New declaration "Canonical Structure id = t : I" to help resolution of equations of the form (proj ?)=a; if proj(e)=a then a is canonically equipped with the remaining fields in e, i.e. ? is instantiated by e

Tactics

- New tactic "ClearBody H" to clear the body of definitions in local context
- New tactic "Assert H := c" for forward reasoning
- Slight improvement in naming strategy for NewInduction/NewDestruct
- Intuition/Tauto do not perform useless unfolding and work up to conversion

Extraction (details in plugins/extraction/CHANGES or documentation)

- Syntax changes: there are no more options inside the extraction commands. New commands for customization and options have been introduced instead.
- More optimizations on extracted code.
- Extraction tests are now embedded in 14 user contributions.

Standard library

- In [Relations], Rstar.v and Newman.v now axiom-free.
- In [Sets], Integers.v now based on nat
- In [Arith], more lemmas in Min.v, new file Max.v, tail-recursive plus and mult added to Plus.v and Mult.v respectively
- New directory [Sorting] with a proof of heapsort (dragged from 6.3.1 lib)
- In [Reals], more lemmas in Rbase.v, new lemmas on square, square root and trigonometric functions (R_sqr.v - Rtrigo.v); a complementary approach and new theorems about continuity and derivability in Ranalysis.v; some properties in plane geometry such as translation, rotation or similarity in Rgeom.v; finite sums and Chasles property in Rsigma.v

Bugs

- Confusion between implicit args of locals and globals of same base name fixed
- Various incompatibilities wrt inference of "?" in V6.3.1 fixed
- Implicits in infix section variables bug fixed
- Known coercions bugs fixed
- Apply "universe anomaly" bug fixed

- NatRing now working
- ”Discriminate 1”, ”Injection 1”, ”Simplify_eq 1” now working
- NewInduction bugs with let-in and recursively dependent hypotheses fixed
- Syntax [x:=t:T]u now allowed as mentioned in documentation
- Bug with recursive inductive types involving let-in fixed
- Known pattern-matching bugs fixed
- Known Cases elimination predicate bugs fixed
- Improved errors messages for pattern-matching and projections
- Better error messages for ill-typed Cases expressions

Incompatibilities

- New naming strategy for NewInduction/NewDestruct may affect 7.1 compatibility
- Extra parentheses may exceptionally be needed in tactic definitions.
- Coq extensions written in Ocaml need to be updated (see dev/changements.txt for a description of the main changes in the interface files of V7.2)
- New behaviour of Intuition/Tauto may exceptionally lead to incompatibilities

2.4.4 Details of changes in 7.3

Language

- Slightly improved compilation of pattern-matching (slight source of incompatibilities)
- Record’s now accept anonymous fields ”_” which does not build projections
- Changes in the allowed elimination sorts for certain class of inductive definitions : an inductive definition without constructors of Sort Prop can be eliminated on sorts Set and Type A ”singleton” inductive definition (one constructor with arguments in the sort Prop like conjunction of two propositions or equality) can be eliminated directly on sort Type (In V7.2, only the sorts Prop and Set were allowed)

Tactics

- New tactic ”Rename x into y” for renaming hypotheses
- New tactics ”Pose x:=u” and ”Pose u” to add definitions to local context
- Pattern now working on partially applied subterms
- Ring no longer applies irreversible congruence laws of mult but better applies congruence laws of plus (slight source of incompatibilities).
- Field now accepts terms to be simplified as arguments (as for Ring). This extension has been also implemented using the toplevel tactic language.
- Intuition does no longer unfold constants except ”<->” and ”~”. It can be parameterized by a tactic. It also can introduce dependent product if needed (source of incompatibilities)
- ”Match Context” now matching more recent hypotheses first and failing only on user errors and Fail tactic (possible source of incompatibilities)
- Tactic Definition’s without arguments now allowed in Coq states
- Better simplification and discrimination made by Inversion (source of incompatibilities)

Bugs

- ”Intros H” now working like ”Intro H” trying first to reduce if not a product
- Forward dependencies in Cases now taken into account
- Known bugs related to Inversion and let-in’s fixed
- Bug unexpected Delta with let-in now fixed

Extraction (details in plugins/extraction/CHANGES or documentation)

- Signatures of extracted terms are now mostly expunged from dummy arguments.
- Haskell extraction is now operational (tested & debugged).

Standard library

- Some additions in [ZArith]: three files (Zcomplements.v, Zpower.v and Zlogarithms.v) moved from plugins/omega in order to be more visible, one Zsgn function, more induction principles (Wf_Z.v and tail of Zcomplements.v), one more general Euclid theorem
- Peano_dec.v and Compare_dec.v now part of Arith.v

Tools

- new option -dump-glob to coqtop to dump globalizations (to be used by the new documentation tool coqdoc; see http://www.lri.fr/~filliatr/coqdoc)

User Contributions

- CongruenceClosure (congruence closure decision procedure) [Pierre Corbineau, ENS Cachan]
- MapleMode (an interface to embed Maple simplification procedures over rational fractions in Coq) [David Delahaye, Micaela Mayero, Chalmers University]
- Presburger: A formalization of Presburger’s algorithm [Laurent Thery, INRIA Sophia Antipolis]
- Chinese has been rewritten using Z from ZArith as datatype ZChinese is the new version, Chinese the obsolete one [Pierre Letouzey, LRI Orsay]

Incompatibilities

- Ring: exceptional incompatibilities (1 above 650 in submitted user contribs, leading to a simplification)
- Intuition: does not unfold any definition except ”<->” and ”~”
- Cases: removal of some extra Cases in configurations of the form ”Cases ... of C _ => ... | _ D => ...” (effects on 2 definitions of submitted user contributions necessitating the removal of now superfluous proof steps in 3 different proofs)
- Match Context, in case of incompatibilities because of a now non trapped error (e.g. Not_found or Failure), use instead tactic Fail to force Match Context trying the next clause
- Inversion: better simplification and discrimination may occasionally lead to less subgoals and/or hypotheses and different naming of hypotheses
- Unification done by Apply/Elim has been changed and may exceptionally lead to incompatible instantiations
- Peano_dec.v and Compare_dec.v parts of Arith.v make Auto more powerful if these files were not already required (1 occurrence of this in submitted user contribs)

Changes in 7.3.1

Bug fixes

- Corrupted Field tactic and Match Context tactic construction fixed
- Checking of names already existing in Assert added (#1386)
- Invalid argument bug in Exact tactic solved (#1387)
- Colliding bound names bug fixed (#1412)
- Wrong non-recursivity test for Record fixed (#1394)
- Out of memory/seg fault bug related to parametric inductive fixed (#1404)
- Setoid_replace/Setoid_rewrite bug wrt "==" fixed

Misc

- Ocaml version >= 3.06 is needed to compile Coq from sources
- Simplification of fresh names creation strategy for Assert, Pose and LetTac (#1402)

2.4.5 Details of changes in 7.4

Symbolic notations

- Introduction of a notion of scope gathering notations in a consistent set; a notation sets has been developed for nat, Z and R (undocumented)
- New command "Notation" for declaring notations simultaneously for parsing and printing (see chap 10 of the reference manual)
- Declarations with only implicit arguments now handled (e.g. the argument of nil can be set implicit; use !nil to refer to nil without arguments)
- "Print Scope sc" and "Locate ntn" allows to know to what expression a notation is bound
- New defensive strategy for printing or not implicit arguments to ensure re-type-checkability of the printed term
- In Grammar command, the only predefined non-terminal entries are ident, global, constr and pattern (e.g. nvar, numarg disappears); the only allowed grammar types are constr and pattern; ast and ast list are no longer supported; some incompatibilities in Grammar: when a syntax is a initial segment of an other one, Grammar does not work, use Notation

Library

- Lemmas in Set from Compare_dec.v (le_lt_dec, ...) and Wf_nat.v (lt_wf_rec, ...) are now transparent. This may be source of incompatibilities.
- Syntactic Definitions Fst, Snd, Ex, All, Ex2, AllT, ExT, ExT2, ProjS1, ProjS2, Error, Value and Except are turned to notations. They now must be applied (incompatibilities only in unrealistic cases).
- More efficient versions of Zmult and times (30% faster)
- Reals: the library is now divided in 6 parts (Rbase, Rfunctions, SeqSeries, Rtrigo, Ranalysis, Integration). New tactics: Sup and RCompute. See Reals.v for details.

Modules

- Beta version, see doc chap 2.5 for commands and chap 5 for theory

Language

- Inductive definitions now accept ">" in constructor types to declare the corresponding constructor as a coercion.
- Idem for assumptions declarations and constants when the type is mentioned.
- The "Coercion" and "Canonical Structure" keywords now accept the same syntax as "Definition", i.e. "hyps :=c (:t)?" or "hyps :t".
- Theorem-like declaration now accepts the syntax "Theorem thm [x:t;...] : u".
- Remark's and Fact's now definitively behave as Theorem and Lemma: when sections are closed, the full name of a Remark or a Fact has no longer a section part (source of incompatibilities)
- Opaque Local's (i.e. built by tactics and ended by Qed), do not survive section closing any longer; as a side-effect, Opaque Local's now appear in the local context of proofs; their body is hidden though (source of incompatibilities); use one of Remark/Fact/Lemma/Theorem instead to simulate the old behaviour of Local (the section part of the name is not kept though)

ML tactic and vernacular commands

- "Grammar tactic" and "Grammar vernac" of type "ast" are no longer supported (only "Grammar tactic simple_tactic" of type "tactic" remains available).
- Concrete syntax for ML written vernacular commands and tactics is now declared at ML level using camlp4 macros TACTIC EXTEND et VERNAC COMMAND EXTEND.
- "Check n c" now "n:Check c", "Eval n ..." now "n:Eval ..."
- `Proof with T` (no documentation)
- SearchAbout id - prints all theorems which contain id in their type

Tactic definitions

- Static globalisation of identifiers and global references (source of incompatibilities, especially, Recursive keyword is required for mutually recursive definitions).
- New evaluation semantics: no more partial evaluation at definition time; evaluation of all Tactic/Meta Definition, even producing terms, expect a proof context to be evaluated (especially "()" is no longer needed).
- Debugger now shows the nesting level and the reasons of failure

Tactics

- Equality tactics (Rewrite, Reflexivity, Symmetry, Transitivity) now understand JM equality
- Simpl and Change now apply to subterms also
- "Simpl f" reduces subterms whose head constant is f
- Double Induction now referring to hypotheses like "Intros until"
- "Inversion" now applies also on quantified hypotheses (naming as for Intros until)
- NewDestruct now accepts terms with missing hypotheses
- NewDestruct and NewInduction now accept user-provided elimination scheme
- NewDestruct and NewInduction now accept user-provided introduction names
- Omega could solve goals such as `~x<y |- x>=y` but failed when the hypothesis was unfolded to `x < y -> False`. This is fixed. In addition, it can also recognize 'False' in the hypothesis and use it to solve the goal.
- Coercions now handled in "with" bindings

- ”Subst x” replaces all occurrences of x by t in the goal and hypotheses when an hypothesis x=t or x:=t or t=x exists
- Fresh names for Assert and Pose now based on collision-avoiding Intro naming strategy (exceptional source of incompatibilities)
- LinearIntuition (no documentation)
- Unfold expects a correct evaluable argument
- Clear expects existing hypotheses

Extraction (See details in plugins/extraction/CHANGES and README):

- An experimental Scheme extraction is provided.
- Concerning Ocaml, extracted code is now ensured to always type-check, thanks to automatic inserting of Obj.magic.
- Experimental extraction of Coq new modules to Ocaml modules.

Proof rendering in natural language

- Export of theories to XML for publishing and rendering purposes now includes proof-trees (see http://www.cs.unibo.it/helm)

Miscellaneous

- Printing Coercion now used through the standard keywords Set/Add, Test, Print
- ”Print Term id” is an alias for ”Print id”
- New switch ”Unset/Set Printing Symbols” to control printing of symbolic notations
- Two new variants of implicit arguments are available
  - `Unset/Set Contextual Implicits` tells to consider implicit also the arguments inferable from the context (e.g. for nil or refl_eq)
  - `Unset/Set Strict Implicits` tells to consider implicit only the arguments that are inferable in any case (i.e. arguments that occurs as argument of rigid constants in the type of the remaining arguments; e.g. the witness of an existential is not strict since it can vanish when applied to a predicate which does not use its argument)

Incompatibilities

- ”Grammar tactic ... : ast” and ”Grammar vernac ... : ast” are no longer supported, use TACTIC EXTEND and VERNAC COMMAND EXTEND on the ML-side instead
- Transparency of le_lt_dec and co (leads to some simplification in proofs; in some cases, incompatibilites is solved by declaring locally opaque the relevant constant)
- Opaque Local do not now survive section closing (rename them into Remark/Lemma/... to get them still surviving the sections; this renaming allows also to solve incompatibilites related to now forbidden calls to the tactic Clear)
- Remark and Fact have no longer (very) long names (use Local instead in case of name conflict)

Bugs

- Improved localisation of errors in Syntactic Definitions
- Induction principle creation failure in presence of let-in fixed (#1459)
- Inversion bugs fixed (#1427 and #1437)
- Omega bug related to Set fixed (#1384)

- Type-checking inefficiency of nested destructuring let-in fixed (#1435)
- Improved handling of let-in during holes resolution phase (#1460)

Efficiency

- Implementation of a memory sharing strategy reducing memory requirements by an average ratio of 3.

CHAPTER

THREE

RECENT CHANGES

3.1 Version 8.11

3.1.1 Summary of changes

The main changes brought by Coq version 8.11 are:

- *Ltac2*, a new tactic language for writing more robust larger scale tactics, with built-in support for datatypes and the multi-goal tactic monad.
- *Primitive floats* are integrated in terms and follow the binary64 format of the IEEE 754 standard, as specified in the `Coq.Float.Floats` library.
- *Cleanups* of the section mechanism, delayed proofs and further restrictions of template polymorphism to fix soundness issues related to universes.
- New *unsafe flags* to disable locally guard, positivity and universe checking. Reliance on these flags is always printed by `Print Assumptions`.
- *Fixed bugs* of `Export` and `Import` that can have a significant impact on user developments (**common source of incompatibility!**).
- New interactive development method based on `vos` *interface files*, allowing to work on a file without recompiling the proof parts of their dependencies.
- New `Arguments` annotation for *bidirectional type inference* configuration for reference (e.g. constants, inductive) applications.
- New *refine attribute* for *`Instance`* can be used instead of the removed `Refine Instance Mode`.
- Generalization of the `under` and `over` *tactics* of SSReflect to arbitrary relations.
- *Revision* of the `Coq.Reals` library, its axiomatisation and instances of the constructive and classical real numbers.

Additionally, while the *`omega`* tactic is not yet deprecated in this version of Coq, it should soon be the case and we already recommend users to switch to *`lia`* in new proof scripts (see also the warning message in the *corresponding chapter*).

The `dev/doc/critical-bugs` file documents the known critical bugs of Coq and affected releases. See the *Changes in 8.11+beta1* section and following sections for the detailed list of changes, including potentially breaking changes marked with **Changed**.

Coq's documentation is available at https://coq.github.io/doc/v8.11/api (documentation of the ML API), https://coq.github.io/doc/v8.11/refman (reference manual), and https://coq.github.io/doc/v8.11/stdlib (documentation of the standard library).

Maxime Dénès, Emilio Jesús Gallego Arias, Gaëtan Gilbert, Michael Soegtrop and Théo Zimmermann worked on maintaining and improving the continuous integration system and package building infrastructure.

The OPAM repository for Coq packages has been maintained by Guillaume Claret, Karl Palmskog, Matthieu Sozeau and Enrico Tassi with contributions from many users. A list of packages is available at https://coq.inria.fr/opam/www/.

The 61 contributors to this version are Michael D. Adams, Guillaume Allais, Helge Bahmann, Langston Barrett, Guillaume Bertholon, Frédéric Besson, Simon Boulier, Michele Caci, Tej Chajed, Arthur Charguéraud, Cyril Cohen, Frédéric Dabrowski, Arthur Azevedo de Amorim, Maxime Dénès, Nikita Eshkeev, Jim Fehrle, Emilio Jesús Gallego Arias, Paolo G. Giarrusso, Gaëtan Gilbert, Georges Gonthier, Jason Gross, Samuel Gruetter, Armaël Guéneau, Hugo Herbelin, Florent Hivert, Jasper Hugunin, Shachar Itzhaky, Jan-Oliver Kaiser, Robbert Krebbers, Vincent Laporte, Olivier Laurent, Samuel Lelièvre, Nicholas Lewycky, Yishuai Li, Jose Fernando Lopez Fernandez, Andreas Lynge, Kenji Maillard, Erik Martin-Dorel, Guillaume Melquiond, Alexandre Moine, Oliver Nash, Wojciech Nawrocki, Antonio Nikishaev, Pierre-Marie Pédrot, Clément Pit-Claudel, Lars Rasmusson, Robert Rand, Talia Ringer, JP Rodi, Pierre Roux, Kazuhiko Sakaguchi, Vincent Semeria, Michael Soegtrop, Matthieu Sozeau, spanjel, Claude Stolze, Enrico Tassi, Laurent Théry, James R. Wilcox, Xia Li-yao, Théo Zimmermann

Many power users helped to improve the design of the new features via the issue and pull request system, the Coq development mailing list, the coq-club@inria.fr mailing list or the Discourse forum[6]. It would be impossible to mention exhaustively the names of everybody who to some extent influenced the development.

Version 8.11 is the sixth release of Coq developed on a time-based development cycle. Its development spanned 3 months from the release of Coq 8.10. Pierre-Marie Pédrot is the release manager and maintainer of this release, assisted by Matthieu Sozeau. This release is the result of 2000+ commits and 300+ PRs merged, closing 75+ issues.

Paris, November 2019,
Matthieu Sozeau for the Coq development team

3.1.2 Changes in 8.11+beta1

Kernel

- **Added:** A built-in support of floating-point arithmetic, allowing one to devise efficient reflection tactics involving numerical computation. Primitive floats are added in the language of terms, following the binary64 format of the IEEE 754 standard, and the related operations are implemented for the different reduction engines of Coq by using the corresponding processor operators in rounding-to-nearest-even. The properties of these operators are axiomatized in the theory `Coq.Floats.FloatAxioms` which is part of the library `Coq.Floats.Floats`. See Section *Primitive Floats* (#9867[7], closes #8276[8], by Guillaume Bertholon, Erik Martin-Dorel, Pierre Roux).
- **Changed:** Internal definitions generated by *`abstract`*-like tactics are now inlined inside universe *`Qed`*-terminated polymorphic definitions, similarly to what happens for their monomorphic counterparts, (#10439[9], by Pierre-Marie Pédrot).

[6] https://coq.discourse.group/
[7] https://github.com/coq/coq/pull/9867
[8] https://github.com/coq/coq/issues/8276
[9] https://github.com/coq/coq/pull/10439

- **Fixed:** Section data is now part of the kernel. Solves a soundness issue in interactive mode where global monomorphic universe constraints would be dropped when forcing a delayed opaque proof inside a polymorphic section. Also relaxes the nesting criterion for sections, as polymorphic sections can now appear inside a monomorphic one (#10664,[10] by Pierre-Marie Pédrot).
- **Changed:** Using `SProp` is now allowed by default, without needing to pass `-allow-sprop` or use *`Allow StrictProp`* (#10811[11], by Gaëtan Gilbert).

Specification language, type inference

- **Added:** Annotation in `Arguments` for bidirectionality hints: it is now possible to tell type inference to use type information from the context once the `n` first arguments of an application are known. The syntax is: `Arguments foo x y & z`. See *`Arguments (bidirectionality hints)`* (#10049[12], by Maxime Dénès with help from Enrico Tassi).
- **Added:** Record fields can be annotated to prevent them from being used as canonical projections; see *Canonical Structures* for details (#10076[13], by Vincent Laporte).
- **Changed:** Require parentheses around nested disjunctive patterns, so that pattern and term syntax are consistent; match branch patterns no longer require parentheses for notation at level 100 or more.

  Warning: Incompatibilities

  – In `match p with (_, (0|1)) => ...` parentheses may no longer be omitted around `0|1`.

  – Notation `(p | q)` now potentially clashes with core pattern syntax, and should be avoided. `-w disj-pattern-notation` flags such *`Notation`*.

  See *Extended pattern matching* for details (#10167[14], by Georges Gonthier).
- **Changed:** *`Function`* always opens a proof when used with a `measure` or `wf` annotation, see *Advanced recursive functions* for the updated documentation (#10215[15], by Enrico Tassi).
- **Changed:** The legacy command *`Add Morphism`* always opens a proof and cannot be used inside a module type. In order to declare a module type parameter that happens to be a morphism, use *`Declare Morphism`*. See *Deprecated syntax and backward incompatibilities* for the updated documentation (#10215[16], by Enrico Tassi).
- **Changed:** The universe polymorphism setting now applies from the opening of a section. In particular, it is not possible anymore to mix polymorphic and monomorphic definitions in a section when there are no variables nor universe constraints defined in this section. This makes the behaviour consistent with the documentation. (#10441[17], by Pierre-Marie Pédrot)
- **Added:** The *`Section`* vernacular command now accepts the ”universes” attribute. In addition to setting the section universe polymorphism, it also locally sets the universe polymorphic option inside the section. (#10441[18], by Pierre-Marie Pédrot)

[10] https://github.com/coq/coq/pull/10664
[11] https://github.com/coq/coq/pull/10811
[12] https://github.com/coq/coq/pull/10049
[13] https://github.com/coq/coq/pull/10076
[14] https://github.com/coq/coq/pull/10167
[15] https://github.com/coq/coq/pull/10215
[16] https://github.com/coq/coq/pull/10215
[17] https://github.com/coq/coq/pull/10441
[18] https://github.com/coq/coq/pull/10441

- **Fixed:** Program Fixpoint now uses ex and sig to make telescopes involving Prop types (#10758[19], by Gaëtan Gilbert, fixing #10757[20] reported by Xavier Leroy).
- **Changed:** Output of the *Print* and *About* commands. Arguments meta-data is now displayed as the corresponding *Arguments* command instead of the human-targeted prose used in previous Coq versions. (#10985[21], by Gaëtan Gilbert).
- **Added:** #[refine] attribute for *Instance*, a more predictable version of the old Refine Instance Mode which unconditionally opens a proof (#10996[22], by Gaëtan Gilbert).
- **Changed:** The unsupported attribute error is now an error-by-default warning, meaning it can be disabled (#10997[23], by Gaëtan Gilbert).
- **Fixed:** Bugs sometimes preventing to define valid (co)fixpoints with implicit arguments in the presence of local definitions, see #3282[24] (#11132[25], by Hugo Herbelin).

Example

The following features an implicit argument after a local definition. It was wrongly rejected.

```
Definition f := fix f (o := true) {n : nat} m {struct m} :=
  match m with 0 => 0 | S m' => f (n:=n+1) m' end.
```

Notations

- **Added:** Numeral Notations now support sorts in the input to printing functions (e.g., numeral notations can be defined for terms containing things like @cons Set nat nil). (#9883[26], by Jason Gross).
- **Added:** The *Notation* and *Infix* commands now support the deprecated attribute (#10180[27], by Maxime Dénès).
- **Deprecated:** The former compat annotation for notations is deprecated, and its semantics changed. It is now made equivalent to using a deprecated attribute, and is no longer connected with the -compat command-line flag (#10180[28], by Maxime Dénès).
- **Changed:** A simplification of parsing rules could cause a slight change of parsing precedences for the very rare users who defined notations with constr at level strictly between 100 and 200 and used these notations on the right-hand side of a cast operator (:, :>, :>>) (#10963[29], by Théo Zimmermann, simplification initially noticed by Jim Fehrle).

Tactics

- **Added:** Syntax injection *term* as [= *intropattern*+] as an alternative to injection *term* as *simple_intropattern*+ using the standard *injection_intropattern* syntax (#9288[30], by Hugo Herbelin).

[19] https://github.com/coq/coq/pull/10758
[20] https://github.com/coq/coq/issues/10757
[21] https://github.com/coq/coq/pull/10985
[22] https://github.com/coq/coq/pull/10996
[23] https://github.com/coq/coq/pull/10997
[24] https://github.com/coq/coq/issues/3282
[25] https://github.com/coq/coq/pull/11132
[26] https://github.com/coq/coq/pull/9883
[27] https://github.com/coq/coq/pull/10180
[28] https://github.com/coq/coq/pull/10180
[29] https://github.com/coq/coq/pull/10963
[30] https://github.com/coq/coq/pull/9288

- **Changed:** Reimplementation of the *zify* tactic. The tactic is more efficient and copes with dependent hypotheses. It can also be extended by redefining the tactic `zify_post_hook`. (#9856[31], fixes #8898[32], #7886[33], #9848[34] and #5155[35], by Frédéric Besson).
- **Changed:** The goal selector tactical `only` now checks that the goal range it is given is valid instead of ignoring goals out of the focus range (#10318[36], by Gaëtan Gilbert).
- **Added:** Flags *Lia Cache*, *Nia Cache* and *Nra Cache*. (#10765[37], by Frédéric Besson, see #10772[38] for use case).
- **Added:** The *zify* tactic is now aware of `Z.to_N`. (#10774[39], grants #9162[40], by Kazuhiko Sakaguchi).
- **Changed:** The *assert_succeeds* and *assert_fails* tactics now only run their tactic argument once, even if it has multiple successes. This prevents blow-up and looping from using multisuccess tactics with *assert_succeeds*. (#10966[41] fixes #10965[42], by Jason Gross).
- **Fixed:** The *assert_succeeds* and *assert_fails* tactics now behave correctly when their tactic fully solves the goal. (#10966[43] fixes #9114[44], by Jason Gross).

Tactic language

- **Added:** Ltac2, a new version of the tactic language Ltac, that doesn't preserve backward compatibility, has been integrated in the main Coq distribution. It is still experimental, but we already recommend users of advanced Ltac to start using it and report bugs or request enhancements. See its documentation in the *dedicated chapter* (#10002[45], plugin authored by Pierre-Marie Pédrot, with contributions by various users, integration by Maxime Dénès, help on integrating / improving the documentation by Théo Zimmermann and Jim Fehrle).
- **Added:** Ltac2 tactic notations with "constr" arguments can specify the interpretation scope for these arguments; see *Notations* for details (#10289[46], by Vincent Laporte).
- **Changed:** White spaces are forbidden in the `&`*ident* syntax for ltac2 references that are described in *Built-in quotations* (#10324[47], fixes #10088[48], authored by Pierre-Marie Pédrot).

SSReflect

- **Added:** Generalize tactics *under* and *over* for any registered relation. More precisely, assume the given context lemma has type `forall f1 f2, .. -> (forall i, R1 (f1 i) (f2 i)) -> R2 f1 f2`. The first step performed by *under* (since Coq 8.10) amounts to calling the tactic *rewrite*, which itself relies on *setoid_rewrite* if need be. So this step was already compatible with a double implication or setoid equality for the conclusion head symbol `R2`. But a further step consists in tagging the generated

[31] https://github.com/coq/coq/pull/9856
[32] https://github.com/coq/coq/issues/8898
[33] https://github.com/coq/coq/issues/7886
[34] https://github.com/coq/coq/issues/9848
[35] https://github.com/coq/coq/issues/5155
[36] https://github.com/coq/coq/pull/10318
[37] https://github.com/coq/coq/pull/10765
[38] https://github.com/coq/coq/issues/10772
[39] https://github.com/coq/coq/pull/10774
[40] https://github.com/coq/coq/issues/9162
[41] https://github.com/coq/coq/pull/10966
[42] https://github.com/coq/coq/issues/10965
[43] https://github.com/coq/coq/pull/10966
[44] https://github.com/coq/coq/issues/9114
[45] https://github.com/coq/coq/pull/10002
[46] https://github.com/coq/coq/pull/10289
[47] https://github.com/coq/coq/pull/10324
[48] https://github.com/coq/coq/issues/10088

subgoal `R1 (f1 i) (?f2 i)` to protect it from unwanted evar instantiation, and get `Under_rel _ R1 (f1 i) (?f2 i)` that is displayed as `'Under[ f1 i ]`. In Coq 8.10, this second (convenience) step was only performed when `R1` was Leibniz' `eq` or `iff`. Now, it is also performed for any relation `R1` which has a `RewriteRelation` instance (a `RelationClasses.Reflexive` instance being also needed so *`over`* can discharge the `'Under[ _ ]` goal by instantiating the hidden evar.) This feature generalizing support for setoid-like relations is enabled as soon as we do both `Require Import ssreflect.` and `Require Setoid.` Finally, a rewrite rule `UnderE` has been added if one wants to "unprotect" the evar, and instantiate it manually with another rule than reflexivity (i.e., without using the *`over`* tactic nor the `over` rewrite rule). See also Section *Rewriting under binders* (#10022[49], by Erik Martin-Dorel, with suggestions and review by Enrico Tassi and Cyril Cohen).

- **Added:** A `void` notation for the standard library empty type (`Empty_set`) (#10932[50], by Arthur Azevedo de Amorim).
- **Added:** Lemma `inj_compr` to `ssr.ssrfun` (#11136[51], by Cyril Cohen).

Commands and options

- **Removed:** Deprecated flag `Refine Instance Mode` (#9530[52], fixes #3632[53], #3890[54] and #4638[55] by Maxime Dénès, review by Gaëtan Gilbert).
- **Changed:** *`Fail`* does not catch critical errors (including "stack overflow") anymore (#10173[56], by Gaëtan Gilbert).
- **Removed:** Undocumented `Instance : !`*`type`* syntax (#10185[57], by Gaëtan Gilbert).
- **Removed:** Deprecated `Show Script` command (#10277[58], by Gaëtan Gilbert).
- **Added:** Unsafe commands to enable/disable guard checking, positivity checking and universes checking (providing a local `-type-in-type`). See *Controlling Typing Flags* (#10291[59] by Simon Boulier).
- **Fixed:** Two bugs in *`Export`*. This can have an impact on the behavior of the *`Import`* command on libraries. `Import A` when `A` imports `B` which exports `C` was importing `C`, whereas *`Import`* is not transitive. Also, after `Import A B`, the import of `B` was sometimes incomplete (#10476[60], by Maxime Dénès).

  Warning: This is a common source of incompatibilities in projects migrating to Coq 8.11.

- **Changed:** Output generated by *`Printing Dependent Evars Line`* flag used by the Prooftree tool in Proof General. (#10489[61], closes #4504[62], #10399[63] and #10400[64], by Jim Fehrle).
- **Added:** Optionally highlight the differences between successive proof steps in the *`Show Proof`* command. Experimental; only available in coqtop and Proof General for now, may be supported in other

[49] https://github.com/coq/coq/pull/10022
[50] https://github.com/coq/coq/pull/10932
[51] https://github.com/coq/coq/pull/11136
[52] https://github.com/coq/coq/pull/9530
[53] https://github.com/coq/coq/issues/3632
[54] https://github.com/coq/coq/issues/3890
[55] https://github.com/coq/coq/issues/4638
[56] https://github.com/coq/coq/pull/10173
[57] https://github.com/coq/coq/pull/10185
[58] https://github.com/coq/coq/pull/10277
[59] https://github.com/coq/coq/pull/10291
[60] https://github.com/coq/coq/pull/10476
[61] https://github.com/coq/coq/pull/10489
[62] https://github.com/coq/coq/issues/4504
[63] https://github.com/coq/coq/issues/10399
[64] https://github.com/coq/coq/issues/10400

IDEs in the future. (#10494[65], by Jim Fehrle).

- **Removed:** Legacy commands AddPath, AddRecPath, and DelPath which were undocumented, broken variants of *Add LoadPath*, *Add Rec LoadPath*, and *Remove LoadPath* (#11187[66], by Maxime Dénès and Théo Zimmermann).

Tools

- **Added:** coqc now provides the ability to generate compiled interfaces. Use coqc -vos foo.v to skip all opaque proofs during the compilation of foo.v, and output a file called foo.vos. This feature is experimental. It enables working on a Coq file without the need to first compile the proofs contained in its dependencies (#8642[67] by Arthur Charguéraud, review by Maxime Dénès and Emilio Gallego).
- **Added:** Command-line options -require-import, -require-export, -require-import-from and -require-export-from, as well as their shorthand, -ri, -re, -refrom and -rifrom. Deprecate confusing command line option -require (#10245[68] by Hugo Herbelin, review by Emilio Gallego).
- **Changed:** Renamed VDFILE from .coqdeps.d to .<CoqMakefile>.d in the coq_makefile utility, where <CoqMakefile> is the name of the output file given by the -o option. In this way two generated makefiles can coexist in the same directory. (#10947[69], by Kazuhiko Sakaguchi).
- **Fixed:** coq_makefile now supports environment variable COQBIN with no ending / character (#11068[70], by Gaëtan Gilbert).

Standard library

- **Changed:** Moved the *auto* hints of the OrderedType module into a new ordered_type database (#9772[71], by Vincent Laporte).
- **Removed:** Deprecated modules Coq.ZArith.Zlogarithm and Coq.ZArith.Zsqrt_compat (#9881[72], by Vincent Laporte).
- **Added:** Module Reals.ConstructiveCauchyReals defines constructive real numbers by Cauchy sequences of rational numbers (#10445[73], by Vincent Semeria, with the help and review of Guillaume Melquiond and Bas Spitters).
- **Added:** New module Reals.ClassicalDedekindReals defines Dedekind real numbers as boolean-valued functions along with 3 logical axioms: limited principle of omniscience, excluded middle of negations, and functional extensionality. The exposed type R in module Reals.Rdefinitions now corresponds to these Dedekind reals, hidden behind an opaque module, which significantly reduces the number of axioms needed (see Reals.Rdefinitions and Reals.Raxioms), while preserving backward compatibility. Classical Dedekind reals are a quotient of constructive reals, which allows to transport many constructive proofs to the classical case (#10827[74], by Vincent Semeria, based on discussions with Guillaume Melquiond, Bas Spitters and Hugo Herbelin, code review by Hugo Herbelin).
- **Added:** New lemmas on combine, filter, nodup, nth, and nth_error functions on lists (#10651[75], and #10731[76], by Oliver Nash).

[65] https://github.com/coq/coq/pull/10494
[66] https://github.com/coq/coq/pull/11187
[67] https://github.com/coq/coq/pull/8642
[68] https://github.com/coq/coq/pull/10245
[69] https://github.com/coq/coq/pull/10947
[70] https://github.com/coq/coq/pull/11068
[71] https://github.com/coq/coq/pull/9772
[72] https://github.com/coq/coq/pull/9811
[73] https://github.com/coq/coq/pull/10445
[74] https://github.com/coq/coq/pull/10827
[75] https://github.com/coq/coq/pull/10651
[76] https://github.com/coq/coq/pull/10731

- **Changed:** The lemma `filter_app` was moved to the `List` module (#10651[77], by Oliver Nash).
- **Added:** Standard equivalence between weak excluded-middle and the classical instance of De Morgan's law, in module `ClassicalFacts` (#10895[78], by Hugo Herbelin).

Infrastructure and dependencies

- **Changed:** Coq now officially supports OCaml 4.08. See `INSTALL` file for details (#10471[79], by Emilio Jesús Gallego Arias).

3.1.3 Changes in 8.11.0

Kernel

- **Changed:** the native compilation (*`native_compute`*) now creates a directory to contain temporary files instead of putting them in the root of the system temporary directory (#11081[80], by Gaëtan Gilbert).
- **Fixed:** #11360[81]. Broken section closing when a template polymorphic inductive type depends on a section variable through its parameters (#11361[82], by Gaëtan Gilbert).
- **Fixed:** The type of `Set+1` would be computed to be itself, leading to a proof of False (#11422[83], by Gaëtan Gilbert).

Specification language, type inference

- **Changed:** Heuristics for universe minimization to `Set`: only minimize flexible universes (#10657[84], by Gaëtan Gilbert with help from Maxime Dénès and Matthieu Sozeau).
- **Fixed:** A dependency was missing when looking for default clauses in the algorithm for printing pattern matching clauses (#11233[85], by Hugo Herbelin, fixing #11231[86], reported by Barry Jay).

Notations

- **Fixed:** *`Print Visibility`* was failing in the presence of only-printing notations (#11276[87], by Hugo Herbelin, fixing #10750[88]).
- **Fixed:** Recursive notations with custom entries were incorrectly parsing `constr` instead of custom grammars (#11311[89] by Maxime Dénès, fixes #9532[90], #9490[91]).

Tactics

- **Changed:** The tactics *`eapply`*, *`refine`* and variants no longer allow shelved goals to be solved by typeclass resolution (#10762[92], by Matthieu Sozeau).

[77] https://github.com/coq/coq/pull/10651
[78] https://github.com/coq/coq/pull/10895
[79] https://github.com/coq/coq/pull/10471
[80] https://github.com/coq/coq/pull/11081
[81] https://github.com/issues/11360
[82] https://github.com/coq/coq/pull/11361
[83] https://github.com/coq/coq/pull/11422
[84] https://github.com/coq/coq/pull/10657
[85] https://github.com/coq/coq/pull/11233
[86] https://github.com/coq/coq/pull/11231
[87] https://github.com/coq/coq/pull/11276
[88] https://github.com/coq/coq/pull/10750
[89] https://github.com/coq/coq/pull/11311
[90] https://github.com/coq/coq/pull/9532
[91] https://github.com/coq/coq/pull/9490
[92] https://github.com/coq/coq/pull/10762

- **Fixed:** The optional string argument to *time* is now properly quoted under *Print Ltac* (#11203[93], fixes #10971[94], by Jason Gross)
- **Fixed:** Efficiency regression of *lia* introduced in 8.10 by PR #9725[95] (#11263[96], fixes #11063[97], and #11242[98], and #11270[99], by Frédéric Besson).
- **Deprecated:** The undocumented `omega with` tactic variant has been deprecated. Using *lia* is the recommended replacement, though the old semantics of `omega with *` can be recovered with `zify; omega` (#11337[100], by Emilio Jesus Gallego Arias).
- **Fixed** For compatibility reasons, in 8.11, *zify* does not support `Z.pow_pos` by default. It can be enabled by explicitly loading the module `ZifyPow` (#11430[101] by Frédéric Besson fixes #11191[102]).

Tactic language

- **Fixed:** Syntax of tactic `cofix ... with ...` was broken since Coq 8.10 (#11241[103], by Hugo Herbelin).

Commands and options

- **Deprecated:** The `-load-ml-source` and `-load-ml-object` command line options have been deprecated; their use was very limited, you can achieve the same by adding object files in the linking step or by using a plugin (#11428[104], by Emilio Jesus Gallego Arias).

Tools

- **Fixed:** `coqtop --version` was broken when called in the middle of an installation process (#11255[105], by Hugo Herbelin, fixing #11254[106]).
- **Deprecated:** The `-quick` command is renamed to `-vio`, for consistency with the new `-vos` and `-vok` flags. Usage of `-quick` is now deprecated (#11280[107], by Arthur Charguéraud).
- **Fixed:** `coq_makefile` does not break when using the `CAMLPKGS` variable together with an unpacked (`mllib`) plugin (#11357[108], by Gaëtan Gilbert).
- **Fixed:** `coqdoc` with option `-g` (Gallina only) now correctly prints commands with attributes (#11394[109], fixes #11353[110], by Karl Palmskog).

CoqIDE

- **Changed:** CoqIDE now uses the GtkSourceView native implementation of the autocomplete mechanism (#11400[111], by Pierre-Marie Pédrot).

Standard library

[93] https://github.com/coq/coq/pull/11203
[94] https://github.com/coq/coq/issues/10971
[95] https://github.com/coq/coq/pull/9725
[96] https://github.com/coq/coq/pull/11263
[97] https://github.com/coq/coq/issues/11063
[98] https://github.com/coq/coq/issues/11242
[99] https://github.com/coq/coq/issues/11270
[100] https://github.com/coq/coq/pull/11337
[101] https://github.com/coq/coq/pull/11430
[102] https://github.com/coq/coq/issues/11191
[103] https://github.com/coq/coq/pull/11241
[104] https://github.com/coq/coq/pull/11428
[105] https://github.com/coq/coq/pull/11255
[106] https://github.com/coq/coq/pull/11254
[107] https://github.com/coq/coq/pull/11280
[108] https://github.com/coq/coq/pull/11357
[109] https://github.com/coq/coq/pull/11394
[110] https://github.com/coq/coq/issues/11353
[111] https://github.com/coq/coq/pull/11400

- **Removed:** Export of module `RList` in `Ranalysis` and `Ranalysis_reg`. Module `RList` is still there but must be imported explicitly where required (#11396[112], by Michael Soegtrop).

Infrastructure and dependencies

- **Added:** Build date can now be overridden by setting the `SOURCE_DATE_EPOCH` environment variable (#11227[113], by Bernhard M. Wiedemann).

3.1.4 Changes in 8.11.1

Kernel

- **Fixed:** Allow more inductive types in `Unset Positivity Checking` mode (#11811[114], by Simon-Boulier).

Notations

- **Fixed:** Bugs in dealing with precedences of notations in custom entries (#11530[115], by Hugo Herbelin, fixing in particular #9517[116], #9519[117], #9521[118], #11331[119]).
- **Added:** In primitive floats, print a warning when parsing a decimal value that is not exactly a binary64 floating-point number. For instance, parsing 0.1 will print a warning whereas parsing 0.5 won't. (#11859[120], by Pierre Roux).

CoqIDE

- **Fixed:** Compiling file paths containing spaces (#10008[121], by snyke7, fixing #11595[122]).

Infrastructure and dependencies

- **Added:** Bump official OCaml support and CI testing to 4.10.0 (#11131[123], #11123[124], #11102[125], by Emilio Jesus Gallego Arias, Jacques-Henri Jourdan, Guillaume Melquiond, and Guillaume Munch-Maccagnoni).

Miscellaneous

- **Fixed:** *`Extraction Implicit`* on the constructor of a record was leading to an anomaly (#11329[126], by Hugo Herbelin, fixes #11114[127]).

3.1.5 Changes in 8.11.2

Kernel

[112] https://github.com/coq/coq/pull/11396
[113] https://github.com/coq/coq/pull/11227
[114] https://github.com/coq/coq/pull/11811
[115] https://github.com/coq/coq/pull/11530
[116] https://github.com/coq/coq/pull/9517
[117] https://github.com/coq/coq/pull/9519
[118] https://github.com/coq/coq/pull/9521
[119] https://github.com/coq/coq/pull/11331
[120] https://github.com/coq/coq/pull/11859
[121] https://github.com/coq/coq/pull/10008
[122] https://github.com/coq/coq/pull/11595
[123] https://github.com/coq/coq/pull/11131
[124] https://github.com/coq/coq/pull/11123
[125] https://github.com/coq/coq/pull/11123
[126] https://github.com/coq/coq/pull/11329
[127] https://github.com/coq/coq/pull/11114

- **Fixed:** Using *Require* inside a section caused an anomaly when closing the section. (#11972[128], by Gaëtan Gilbert, fixing #11783[129], reported by Attila Boros).

Tactics

- **Fixed:** Anomaly with induction schemes whose conclusion is not normalized (#12116[130], by Hugo Herbelin; fixes #12045[131])
- **Fixed:** Loss of location of some tactic errors (#12223[132], by Hugo Herbelin; fixes #12152[133] and #12255[134]).

Commands and options

- **Changed:** Ignore -native-compiler option when built without native compute support. (#12070[135], by Pierre Roux).

CoqIDE

- **Changed:** CoqIDE now uses native window frames by default on Windows. The GTK window frames can be restored by setting the `GTK_CSD` environment variable to `1` (#12060[136], fixes #11080[137], by Attila Gáspár).
- **Fixed:** New patch presumably fixing the random Coq 8.11 segfault issue with CoqIDE completion (#12068[138], by Hugo Herbelin, presumably fixing #11943[139]).
- **Fixed:** Highlighting style consistently applied to all three buffers of CoqIDE (#12106[140], by Hugo Herbelin; fixes #11506[141]).

3.2 Version 8.10

3.2.1 Summary of changes

Coq version 8.10 contains two major new features: support for a native fixed-precision integer type and a new sort `SProp` of strict propositions. It is also the result of refinements and stabilization of previous features, deprecations or removals of deprecated features, cleanups of the internals of the system and API, and many documentation improvements. This release includes many user-visible changes, including deprecations that are documented in the next subsection, and new features that are documented in the reference manual. Here are the most important user-visible changes:

- Kernel:
    - A notion of primitive object was added to the calculus. Its first instance is primitive cyclic unsigned integers, axiomatized in module `UInt63`. See Section *Primitive Integers*. The `Coq.Numbers.Cyclic.Int31` library is deprecated (#6914[142], by Maxime Dénès, Benjamin Grégoire and Vincent Laporte, with help and reviews from many others).

[128] https://github.com/coq/coq/pull/11972
[129] https://github.com/coq/coq/issues/11783
[130] https://github.com/coq/coq/pull/12116
[131] https://github.com/coq/coq/pull/12045
[132] https://github.com/coq/coq/pull/12223
[133] https://github.com/coq/coq/pull/12152
[134] https://github.com/coq/coq/pull/12255
[135] https://github.com/coq/coq/pull/12070
[136] https://github.com/coq/coq/pull/12060
[137] https://github.com/coq/coq/issues/11080
[138] https://github.com/coq/coq/pull/12068
[139] https://github.com/coq/coq/pull/11943
[140] https://github.com/coq/coq/pull/12106
[141] https://github.com/coq/coq/pull/11506
[142] https://github.com/coq/coq/pull/6914

  - The SProp sort of definitionally proof-irrelevant propositions was introduced. SProp allows to mark proof terms as irrelevant for conversion, and is treated like Prop during extraction. It is enabled using the `-allow-sprop` command-line flag or the *`Allow StrictProp`* flag. See Chapter *SProp (proof irrelevant propositions)* (#8817[143], by Gaëtan Gilbert).
  - The unfolding heuristic in termination checking was made more complete, allowing more constants to be unfolded to discover valid recursive calls. Performance regression may occur in Fixpoint declarations without an explicit `{struct}` annotation, since guessing the decreasing argument can now be more expensive (#9602[144], by Enrico Tassi).
- Universes:
  - Added *`Print Universes Subgraph`* variant of *`Print Universes`*. Try for instance `Print Universes Subgraph(sigT2.u1 sigT_of_sigT2.u1 projT3_eq.u1).` (#8451[145], by Gaëtan Gilbert).
  - Added private universes for opaque polymorphic constants, see the documentation for the *`Private Polymorphic Universes`* flag, and unset it to get the previous behaviour (#8850[146], by Gaëtan Gilbert).
- Notations:
  - New command *`String Notation`* to register string syntax for custom inductive types (#8965[147], by Jason Gross).
  - Experimental: *Numeral Notations* now parse decimal constants such as `1.02e+01` or `10.2`. Parsers added for `Q` and `R`. In the rare case when such numeral notations were used in a development along with `Q` or `R`, they may have to be removed or disambiguated through explicit scope annotations (#8764[148], by Pierre Roux).
- Ltac backtraces can be turned on using the *`Ltac Backtrace`* flag, which is off by default (#9142[149], fixes #7769[150] and #7385[151], by Pierre-Marie Pédrot).
- The tactics *`lia`*, *`nia`*, *`lra`*, *`nra`* are now using a novel Simplex-based proof engine. In case of regression, unset *`Simplex`* to get the venerable Fourier-based engine (#8457[152], by Fréderic Besson).
- SSReflect:
  - New intro patterns:
    * temporary introduction: `=> +`
    * block introduction: `=> [^ prefix ] [^~ suffix ]`
    * fast introduction: `=> >`
    * tactics as views: `=> /ltac:mytac`
    * replace hypothesis: `=> {}H`

      See Section *Introduction in the context* (#6705[153], by Enrico Tassi, with help from Maxime Dénès, ideas coming from various users).

[143] https://github.com/coq/coq/pull/8817
[144] https://github.com/coq/coq/pull/9602
[145] https://github.com/coq/coq/pull/8451
[146] https://github.com/coq/coq/pull/8850
[147] https://github.com/coq/coq/pull/8965
[148] https://github.com/coq/coq/pull/8764
[149] https://github.com/coq/coq/pull/9142
[150] https://github.com/coq/coq/issues/7769
[151] https://github.com/coq/coq/issues/7385
[152] https://github.com/coq/coq/pull/8457
[153] https://github.com/coq/coq/pull/6705

  - New tactic *under* to rewrite under binders, given an extensionality lemma:
    - interactive mode: `under` *term*, associated terminator: *over*
    - one-liner mode: `under` *term* `do [`*tactic* `| ...]`

    It can take occurrence switches, contextual patterns, and intro patterns: `under {2}[in RHS]eq_big => [i|i ?]` (#9651[154], by Erik Martin-Dorel and Enrico Tassi).

- *Combined Scheme* now works when inductive schemes are generated in sort `Type`. It used to be limited to sort `Prop` (#7634[155], by Théo Winterhalter).
- A new registration mechanism for reference from ML code to Coq constructs has been added (#186[156], by Emilio Jesús Gallego Arias, Maxime Dénès and Vincent Laporte).
- CoqIDE:
  - CoqIDE now depends on gtk+3 and lablgtk3 instead of gtk+2 and lablgtk2. The INSTALL file available in the Coq sources has been updated to list the new dependencies (#9279[157], by Hugo Herbelin, with help from Jacques Garrigue, Emilio Jesús Gallego Arias, Michael Sogetrop and Vincent Laporte).
  - Smart input for Unicode characters. For example, typing `\alpha` then `Shift+Space` will insert the greek letter alpha. A larger number of default bindings are provided, following the latex naming convention. Bindings can be customized, either globally, or on a per-project basis. See Section *Bindings for input of Unicode symbols* for details (#8560[158], by Arthur Charguéraud).
- Infrastructure and dependencies:
  - Coq 8.10 requires OCaml >= 4.05.0, bumped from 4.02.3 See the `INSTALL` file for more information on dependencies (#7522[159], by Emilio Jesús Gallego Arías).
  - Coq 8.10 doesn't need Camlp5 to build anymore. It now includes a fork of the core parsing library that Coq uses, which is a small subset of the whole Camlp5 distribution. In particular, this subset doesn't depend on the OCaml AST, allowing easier compilation and testing on experimental OCaml versions. Coq also ships a new parser `coqpp` that plugin authors must switch to (#7902[160], #7979[161], #8161[162], #8667[163], and #8945[164], by Pierre-Marie Pédrot and Emilio Jesús Gallego Arias).

    The Coq developers would like to thank Daniel de Rauglaudre for many years of continued support.
  - Coq now supports building with Dune, in addition to the traditional Makefile which is scheduled for deprecation (#6857[165], by Emilio Jesús Gallego Arias, with help from Rudi Grinberg).

    Experimental support for building Coq projects has been integrated in Dune at the same time, providing an improved experience[166] for plugin developers. We thank the Dune team for their work supporting Coq.

[154] https://github.com/coq/coq/pull/9651
[155] https://github.com/coq/coq/pull/7634
[156] https://github.com/coq/coq/pull/186
[157] https://github.com/coq/coq/pull/9279
[158] https://github.com/coq/coq/pull/8560
[159] https://github.com/coq/coq/pull/7522
[160] https://github.com/coq/coq/pull/7902
[161] https://github.com/coq/coq/pull/7979
[162] https://github.com/coq/coq/pull/8161
[163] https://github.com/coq/coq/pull/8667
[164] https://github.com/coq/coq/pull/8945
[165] https://github.com/coq/coq/pull/6857
[166] https://coq.discourse.group/t/a-guide-to-building-your-coq-libraries-and-plugins-with-dune/

Version 8.10 also comes with a bunch of smaller-scale changes and improvements regarding the different components of the system, including many additions to the standard library (see the next subsection for details).

On the implementation side, the `dev/doc/changes.md` file documents the numerous changes to the implementation and improvements of interfaces. The file provides guidelines on porting a plugin to the new version and a plugin development tutorial originally made by Yves Bertot is now in `doc/plugin_tutorial`. The `dev/doc/critical-bugs` file documents the known critical bugs of Coq and affected releases.

The efficiency of the whole system has seen improvements thanks to contributions from Gaëtan Gilbert, Pierre-Marie Pédrot, and Maxime Dénès.

Maxime Dénès, Emilio Jesús Gallego Arias, Gaëtan Gilbert, Michael Soegtrop, Théo Zimmermann worked on maintaining and improving the continuous integration system and package building infrastructure. Coq is now continuously tested against OCaml trunk, in addition to the oldest supported and latest OCaml releases.

Coq's documentation for the development branch is now deployed continuously at https://coq.github.io/doc/master/api (documentation of the ML API), https://coq.github.io/doc/master/refman (reference manual), and https://coq.github.io/doc/master/stdlib (documentation of the standard library). Similar links exist for the `v8.10` branch.

The OPAM repository for Coq packages has been maintained by Guillaume Melquiond, Matthieu Sozeau, Enrico Tassi (who migrated it to opam 2) with contributions from many users. A list of packages is available at https://coq.inria.fr/opam/www/.

The 61 contributors to this version are Tanaka Akira, Benjamin Barenblat, Yves Bertot, Frédéric Besson, Lasse Blaauwbroek, Martin Bodin, Joachim Breitner, Tej Chajed, Frédéric Chapoton, Arthur Charguéraud, Cyril Cohen, Lukasz Czajka, David A. Dalrymple, Christian Doczkal, Maxime Dénès, Andres Erbsen, Jim Fehrle, Emilio Jesus Gallego Arias, Gaëtan Gilbert, Matěj Grabovský, Simon Gregersen, Jason Gross, Samuel Gruetter, Hugo Herbelin, Jasper Hugunin, Mirai Ikebuchi, Chantal Keller, Matej Košík, Sam Pablo Kuper, Vincent Laporte, Olivier Laurent, Larry Darryl Lee Jr, Nick Lewycky, Yao Li, Yishuai Li, Assia Mahboubi, Simon Marechal, Erik Martin-Dorel, Thierry Martinez, Guillaume Melquiond, Kayla Ngan, Karl Palmskog, Pierre-Marie Pédrot, Clément Pit-Claudel, Pierre Roux, Kazuhiko Sakaguchi, Ryan Scott, Vincent Semeria, Gan Shen, Michael Soegtrop, Matthieu Sozeau, Enrico Tassi, Laurent Théry, Kamil Trzciński, whitequark, Théo Winterhalter, Xia Li-yao, Beta Ziliani and Théo Zimmermann.

Many power users helped to improve the design of the new features via the issue and pull request system, the Coq development mailing list, the coq-club@inria.fr mailing list or the new Discourse forum. It would be impossible to mention exhaustively the names of everybody who to some extent influenced the development.

Version 8.10 is the fifth release of Coq developed on a time-based development cycle. Its development spanned 6 months from the release of Coq 8.9. Vincent Laporte is the release manager and maintainer of this release. This release is the result of ~2500 commits and ~650 PRs merged, closing 150+ issues.

Santiago de Chile, April 2019,
Matthieu Sozeau for the Coq development team

3.2.2 Other changes in 8.10+beta1

- Command-line tools and options:
  - The use of `coqtop` as a compiler has been deprecated, in favor of `coqc`. Consequently option `-compile` will stop to be accepted in the next release. `coqtop` is now reserved to interactive use

(#9095[167], by Emilio Jesús Gallego Arias).

- New option `-topfile filename`, which will set the current module name (*à la* `-top`) based on the filename passed, taking into account the proper `-R`/`-Q` options. For example, given `-R Foo foolib` using `-topfile foolib/bar.v` will set the module name to `Foo.Bar`. CoqIDE now properly sets the module name for a given file based on its path (#8991[168], closes #8989[169], by Gaëtan Gilbert).
- Experimental: Coq flags and options can now be set on the command-line, e.g. `-set "Universe Polymorphism=true"` (#9876[170], by Gaëtan Gilbert).
- The `-native-compiler` flag of `coqc` and `coqtop` now takes an argument which can have three values:

  * `no` disables native_compute
  * `yes` enables native_compute and precompiles `.v` files to native code
  * `ondemand` enables native_compute but compiles code only when `native_compute` is called

  The default value is `ondemand`. Note that this flag now has priority over the configure flag of the same name.

  A new `-bytecode-compiler` flag for `coqc` and `coqtop` controls whether conversion can use the VM. The default value is `yes`.

  (#8870[171], by Maxime Dénès)
- The pretty timing diff scripts (flag `TIMING=1` to a `coq_makefile`-made `Makefile`, also `tools/make-both-single-timing-files.py`, `tools/make-both-time-files.py`, and `tools/make-one-time-file.py`) now correctly support non-UTF-8 characters in the output of `coqc` / `make` as well as printing to stdout, on both python2 and python3 (#9872[172], closes #9767[173] and #9705[174], by Jason Gross)
- coq_makefile's install target now errors if any file to install is missing (#9906[175], by Gaëtan Gilbert).
- Preferences from `coqide.keys` are no longer overridden by modifiers preferences in `coqiderc` (#10014[176], by Hugo Herbelin).

- Specification language, type inference:
  - Fixing a missing check in interpreting instances of existential variables that are bound to local definitions. Might exceptionally induce an overhead if the cost of checking the conversion of the corresponding definitions is additionally high (#8217[177], closes #8215[178], by Hugo Herbelin).
  - A few improvements in inference of the return clause of `match` that can exceptionally introduce incompatibilities. This can be solved by writing an explicit `return` clause, sometimes even simply an explicit `return _` clause (#262[179], by Hugo Herbelin).

[167] https://github.com/coq/coq/pull/9095
[168] https://github.com/coq/coq/pull/8991
[169] https://github.com/coq/coq/issues/8989
[170] https://github.com/coq/coq/pull/9876
[171] https://github.com/coq/coq/pull/8870
[172] https://github.com/coq/coq/pull/9872
[173] https://github.com/coq/coq/issues/9767
[174] https://github.com/coq/coq/issues/9705
[175] https://github.com/coq/coq/pull/9906
[176] https://github.com/coq/coq/pull/10014
[177] https://github.com/coq/coq/pull/8217
[178] https://github.com/coq/coq/issues/8215
[179] https://github.com/coq/coq/pull/262

  - Using non-projection values with the projection syntax is not allowed. For instance `0.(S)` is not a valid way to write `S 0`. Projections from non-primitive (emulated) records are allowed with warning ”nonprimitive-projection-syntax” (#8829[180], by Gaëtan Gilbert).
  - An option and attributes to control the automatic decision to declare an inductive type as template polymorphic were added. Warning ”auto-template” (off by default) can trigger when an inductive is automatically declared template polymorphic without the attribute.

    Inductive types declared by Funind will never be template polymorphic.

    (#8488[181], by Gaëtan Gilbert)

- Notations:
  - New command *`Declare Scope`* to explicitly declare a scope name before any use of it. Implicit declaration of a scope at the time of *`Bind Scope`*, *`Delimit Scope`*, *`Undelimit Scope`*, or *`Notation`* is deprecated (#7135[182], by Hugo Herbelin).
  - Various bugs have been fixed (e.g. #9214[183] on removing spurious parentheses on abbreviations shortening a strict prefix of an application, by Hugo Herbelin).
  - *`Numeral Notation`* now support inductive types in the input to printing functions (e.g., numeral notations can be defined for terms containing things like `@cons nat O O`), and parsing functions now fully normalize terms including parameters of constructors (so that, e.g., a numeral notation whose parsing function outputs a proof of `Nat.gcd x y = 1` will no longer fail to parse due to containing the constant `Nat.gcd` in the parameter-argument of `eq_refl`) (#9874[184], closes #9840[185] and #9844[186], by Jason Gross).
  - Deprecated compatibility notations have actually been removed. Uses of these notations are generally easy to fix thanks to the hint contained in the deprecation warning emitted by Coq 8.8 and 8.9. For projects that require more than a handful of such fixes, there is a script[187] that will do it automatically, using the output of `coqc` (#8638[188], by Jason Gross).
  - Allow inspecting custom grammar entries by *`Print Custom Grammar`* (#10061[189], fixes #9681[190], by Jasper Hugunin, review by Pierre-Marie Pédrot and Hugo Herbelin).
- The quote plugin[191] was removed. If some users are interested in maintaining this plugin externally, the Coq development team can provide assistance for extracting the plugin and setting up a new repository (#7894[192], by Maxime Dénès).
- Ltac:
  - Tactic names are no longer allowed to clash, even if they are not defined in the same section. For example, the following is no longer accepted: `Ltac foo := idtac. Section S. Ltac foo := fail. End S.` (#8555[193], by Maxime Dénès).
  - Names of existential variables occurring in Ltac functions (e.g. `?[n]` or `?n` in terms - not in patterns) are now interpreted the same way as other variable names occurring in Ltac functions

[180] https://github.com/coq/coq/pull/8829
[181] https://github.com/coq/coq/pull/8488
[182] https://github.com/coq/coq/pull/7135
[183] https://github.com/coq/coq/pull/9214
[184] https://github.com/coq/coq/pull/9840
[185] https://github.com/coq/coq/issues/9840
[186] https://github.com/coq/coq/issues/9844
[187] https://gist.github.com/JasonGross/9770653967de3679d131c59d42de6d17#file-replace-notations-py
[188] https://github.com/coq/coq/pull/8638
[189] https://github.com/coq/coq/pull/10061
[190] http://github.com/coq/coq/pull/9681
[191] https://coq.inria.fr/distrib/V8.9.0/refman/proof-engine/detailed-tactic-examples.html#quote
[192] https://github.com/coq/coq/pull/7894
[193] https://github.com/coq/coq/pull/8555

(#7309[194], by Hugo Herbelin).

- Tactics:
  - Removed the deprecated `romega` tactic (#8419[195], by Maxime Dénès and Vincent Laporte).
  - Hint declaration and removal should now specify a database (e.g. `Hint Resolve foo : database`). When the database name is omitted, the hint is added to the `core` database (as previously), but a deprecation warning is emitted (#8987[196], by Maxime Dénès).
  - There are now tactics in `PreOmega.v` called `Z.div_mod_to_equations`, `Z.quot_rem_to_equations`, and `Z.to_euclidean_division_equations` (which combines the `div_mod` and `quot_rem` variants) which allow *lia*, *nia*, etc to support `Z.div` and `Z.modulo` (`Z.quot` and `Z.rem`, respectively), by posing the specifying equation for `Z.div` and `Z.modulo` before replacing them with atoms (#8062[197], by Jason Gross).
  - The syntax of the *autoapply* tactic was fixed to conform with preexisting documentation: it now takes a `with` clause instead of a `using` clause (#9524[198], closes #7632[199], by Théo Zimmermann).
  - Modes are now taken into account by *typeclasses eauto* for local hypotheses (#9996[200], fixes #5752[201], by Maxime Dénès, review by Pierre-Marie Pédrot).
  - New variant *change_no_check* of *change*, usable as a documented replacement of *convert_concl_no_check* (#10012[202], #10017[203], #10053[204], and #10059[205], by Hugo Herbelin and Paolo G. Giarrusso).
  - The simplified value returned by *field_simplify* is not always a fraction anymore. When the denominator is `1`, it returns `x` while previously it was returning `x/1`. This change could break codes that were post-processing application of *field_simplify* to get rid of these `x/1` (#9854[206], by Laurent Théry, with help from Michael Soegtrop, Maxime Dénès, and Vincent Laporte).
- SSReflect:
  - Clear discipline made consistent across the entire proof language. Whenever a clear switch `{x..}` comes immediately before an existing proof context entry (used as a view, as a rewrite rule or as name for a new context entry) then such entry is cleared too.

    E.g. The following sentences are elaborated as follows (when H is an existing proof context entry):

    * `=> {x..} H` -> `=> {x..H} H`
    * `=> {x..} /H` -> `=> /v {x..H}`
    * `rewrite {x..} H` -> `rewrite E {x..H}`

    (#9341[207], by Enrico Tassi).
  - `inE` now expands `y in r x` when `r` is a `simpl_rel`. New `{pred T}` notation for a `pred T` alias in the `pred_sort` coercion class, simplified `predType` interface: `pred_class` and `mkPredType` deprecated, `{pred T}` and `PredType` should be used instead. `if c return t then ...` now

[194] https://github.com/coq/coq/pull/7309
[195] https://github.com/coq/coq/pull/8419
[196] https://github.com/coq/coq/pull/8987
[197] https://github.com/coq/coq/pull/8062
[198] https://github.com/coq/coq/pull/9524
[199] https://github.com/coq/coq/issues/7632
[200] https://github.com/coq/coq/pull/9996
[201] https://github.com/coq/coq/issues/5752
[202] https://github.com/coq/coq/pull/10012
[203] https://github.com/coq/coq/pull/10017
[204] https://github.com/coq/coq/pull/10053
[205] https://github.com/coq/coq/pull/10059
[206] https://github.com/coq/coq/pull/9854
[207] https://github.com/coq/coq/pull/9341

expects `c` to be a variable bound in `t`. New `nonPropType` interface matching types that do __not__ have sort `Prop`. New `relpre R f` definition for the preimage of a relation R under f (#9995[208], by Georges Gonthier).

- Vernacular commands:
  - Binders for an *Instance* now act more like binders for a *Theorem*. Names may not be repeated, and may not overlap with section variable names (#8820[209], closes #8791[210], by Jasper Hugunin).
  - Removed the deprecated `Implicit Tactic` family of commands (#8779[211], by Pierre-Marie Pédrot).
  - The `Automatic Introduction` option has been removed and is now the default (#9001[212], by Emilio Jesús Gallego Arias).
  - `Arguments` now accepts names for arguments provided with `extra_scopes` (#9117[213], by Maxime Dénès).
  - The naming scheme for anonymous binders in a `Theorem` has changed to avoid conflicts with explicitly named binders (#9160[214], closes #8819[215], by Jasper Hugunin).
  - Computation of implicit arguments now properly handles local definitions in the binders for an `Instance`, and can be mixed with implicit binders `{x : T}` (#9307[216], closes #9300[217], by Jasper Hugunin).
  - *Declare Instance* now requires an instance name.

    The flag `Refine Instance Mode` has been turned off by default, meaning that *Instance* no longer opens a proof when a body is provided. The flag has been deprecated and will be removed in the next version.

    (#9270[218], and #9825[219], by Maxime Dénès)
  - Command *Instance*, when no body is provided, now always opens a proof. This is a breaking change, as instance of `Instance` *ident₁* `:` *ident₂*`.` where *ident₂* is a trivial class will have to be changed into `Instance` *ident₁* `:` *ident₂* `:= {}.` or `Instance` *ident₁* `:` *ident₂*`. Proof. Qed.` (#9274[220], by Maxime Dénès).
  - The flag *Program Mode* now means that the `Program` attribute is enabled for all commands that support it. In particular, it does not have any effect on tactics anymore. May cause some incompatibilities (#9410[221], by Maxime Dénès).
  - The algorithm computing implicit arguments now behaves uniformly for primitive projection and application nodes (#9509[222], closes #9508[223], by Pierre-Marie Pédrot).

[208] https://github.com/coq/coq/pull/9995
[209] https://github.com/coq/coq/pull/8820
[210] https://github.com/coq/coq/issues/8791
[211] https://github.com/coq/coq/pull/8779
[212] https://github.com/coq/coq/pull/9001
[213] https://github.com/coq/coq/pull/9117
[214] https://github.com/coq/coq/pull/9160
[215] https://github.com/coq/coq/issues/8819
[216] https://github.com/coq/coq/pull/9307
[217] https://github.com/coq/coq/issues/9300
[218] https://github.com/coq/coq/pull/9270
[219] https://github.com/coq/coq/pull/9825
[220] https://github.com/coq/coq/pull/9274
[221] https://github.com/coq/coq/pull/9410
[222] https://github.com/coq/coq/pull/9509
[223] https://github.com/coq/coq/issues/9508

  - *Hypotheses* and *Variables* can now take implicit binders inside sections (#9364[224], closes #9363[225], by Jasper Hugunin).
  - Removed deprecated option `Automatic Coercions Import` (#8094[226], by Maxime Dénès).
  - The `Show Script` command has been deprecated (#9829[227], by Vincent Laporte).
  - *Coercion* does not warn ambiguous paths which are obviously convertible with existing ones. The ambiguous paths messages have been turned to warnings, thus now they could appear in the output of `coqc`. The convertibility checking procedure for coercion paths is complete for paths consisting of coercions satisfying the uniform inheritance condition, but some coercion paths could be reported as ambiguous even if they are convertible with existing ones when they have coercions that don't satisfy the uniform inheritance condition (#9743[228], closes #3219[229], by Kazuhiko Sakaguchi).
  - A new flag *Fast Name Printing* has been introduced. It changes the algorithm used for allocating bound variable names for a faster but less clever one (#9078[230], by Pierre-Marie Pédrot).
  - Option `Typeclasses Axioms Are Instances` (compatibility option introduced in the previous version) is deprecated. Use *Declare Instance* for axioms which should be instances (#8920[231], by Gaëtan Gilbert).
  - Removed option `Printing Primitive Projection Compatibility` (#9306[232], by Gaëtan Gilbert).

- Standard Library:
  - Added `Bvector.BVeq` that decides whether two `Bvector`s are equal. Added notations for `BVxor`, `BVand`, `BVor`, `BVeq` and `BVneg` (#8171[233], by Yishuai Li).
  - Added `ByteVector` type that can convert to and from `string` (#8365[234], by Yishuai Li).
  - Added lemmas about monotonicity of `N.double` and `N.succ_double`, and about the upper bound of number represented by a vector. Allowed implicit vector length argument in `Ndigits.Bv2N` (#8815[235], by Yishuai Li).
  - The prelude used to be automatically Exported and is now only Imported. This should be relevant only when importing files which don't use `-noinit` into files which do (#9013[236], by Gaëtan Gilbert).
  - Added `Coq.Structures.OrderedTypeEx.String_as_OT` to make strings an ordered type, using lexical order (#7221[237], by Li Yao).
  - Added lemmas about `Z.testbit`, `Z.ones`, and `Z.modulo` (#9425[238], by Andres Erbsen).
  - Moved the `auto` hints of the `FSet` library into a new `fset` database (#9725[239], by Frédéric Besson).

[224] https://github.com/coq/coq/pull/9364
[225] https://github.com/coq/coq/issues/9363
[226] https://github.com/coq/coq/pull/8094
[227] https://github.com/coq/coq/pull/9829
[228] https://github.com/coq/coq/pull/9743
[229] https://github.com/coq/coq/issues/3219
[230] https://github.com/coq/coq/pull/9078
[231] https://github.com/coq/coq/pull/8920
[232] https://github.com/coq/coq/pull/9306
[233] https://github.com/coq/coq/pull/8171
[234] https://github.com/coq/coq/pull/8365
[235] https://github.com/coq/coq/pull/8815
[236] https://github.com/coq/coq/pull/9013
[237] https://github.com/coq/coq/pull/7221
[238] https://github.com/coq/coq/pull/9425
[239] https://github.com/coq/coq/pull/9725

  - Added Coq.Structures.EqualitiesFacts.PairUsualDecidableTypeFull (#9984[240], by Jean-Christophe Léchenet and Oliver Nash).

- Some error messages that show problems with a pair of non-matching values will now highlight the differences (#8669[241], by Jim Fehrle).
- Changelog has been moved from a specific file CHANGES.md to the reference manual; former Credits chapter of the reference manual has been split in two parts: a History chapter which was enriched with additional historical information about Coq versions 1 to 5, and a Changes chapter which was enriched with the content formerly in CHANGES.md and COMPATIBILITY (#9133[242], #9668[243], #9939[244], #9964[245], and #10085[246], by Théo Zimmermann, with help and ideas from Emilio Jesús Gallego Arias, Gaëtan Gilbert, Clément Pit-Claudel, Matthieu Sozeau, and Enrico Tassi).

3.2.3 Changes in 8.10+beta2

Many bug fixes and documentation improvements, in particular:

Tactics

- Make the *discriminate* tactic work together with *Universe Polymorphism* and equality in Type. This, in particular, makes *discriminate* compatible with the HoTT library https://github.com/HoTT/HoTT (#10205[247], by Andreas Lynge, review by Pierre-Marie Pédrot and Matthieu Sozeau).

SSReflect

- Make the case E: t tactic work together with *Universe Polymorphism* and equality in Type. This makes *case* compatible with the HoTT library https://github.com/HoTT/HoTT (#10302[248], fixes #10301[249], by Andreas Lynge, review by Enrico Tassi)
- Make the rewrite /t tactic work together with *Universe Polymorphism*. This makes *rewrite* compatible with the HoTT library https://github.com/HoTT/HoTT (#10305[250], fixes #9336[251], by Andreas Lynge, review by Enrico Tassi)

CoqIDE

- Fix CoqIDE instability on Windows after the update to gtk3 (#10360[252], by Michael Soegtrop, closes #9885[253]).

Miscellaneous

- Proof General can now display Coq-generated diffs between proof steps in color (#10019[254] and (in Proof General) #421[255], by Jim Fehrle).

[240] https://github.com/coq/coq/pull/9984
[241] https://github.com/coq/coq/pull/8669
[242] https://github.com/coq/coq/pull/9133
[243] https://github.com/coq/coq/pull/9668
[244] https://github.com/coq/coq/pull/9939
[245] https://github.com/coq/coq/pull/9964
[246] https://github.com/coq/coq/pull/10085
[247] https://github.com/coq/coq/pull/10205
[248] https://github.com/coq/coq/pull/10302
[249] https://github.com/coq/coq/issues/10301
[250] https://github.com/coq/coq/pull/10305
[251] https://github.com/coq/coq/issues/9336
[252] https://github.com/coq/coq/pull/10360
[253] https://github.com/coq/coq/issues/9885
[254] https://github.com/coq/coq/pull/10019
[255] https://github.com/ProofGeneral/PG/pull/421

3.2.4 Changes in 8.10+beta3

Kernel

- Fix soundness issue with template polymorphism (#9294[256]).

  Declarations of template-polymorphic inductive types ignored the provenance of the universes they were abstracting on and did not detect if they should be greater or equal to `Set` in general. Previous universes and universes introduced by the inductive definition could have constraints that prevented their instantiation with e.g. `Prop`, resulting in unsound instantiations later. The implemented fix only allows abstraction over universes introduced by the inductive declaration, and properly records all their constraints by making them by default only >= `Prop`. It is also checked that a template polymorphic inductive actually is polymorphic on at least one universe.

  This prevents inductive declarations in sections to be universe polymorphic over section parameters. For a backward compatible fix, simply hoist the inductive definition out of the section. An alternative is to declare the inductive as universe-polymorphic and cumulative in a universe-polymorphic section: all universes and constraints will be properly gathered in this case. See *Template polymorphism* for a detailed exposition of the rules governing template-polymorphic types.

  To help users incrementally fix this issue, a command line option `-no-template-check` and a global flag *`Template Check`* are available to selectively disable the new check. Use at your own risk.

  (#9918[257], by Matthieu Sozeau and Maxime Dénès).

User messages

- Improve the ambiguous paths warning to indicate which path is ambiguous with new one (#10336[258], closes #3219[259], by Kazuhiko Sakaguchi).

Extraction

- Fix extraction to OCaml of primitive machine integers; see *Primitive Integers* (#10430[260], fixes #10361[261], by Vincent Laporte).
- Fix a printing bug of OCaml extraction on dependent record projections, which produced improper `assert false`. This change makes the OCaml extractor internally inline record projections by default; thus the monolithic OCaml extraction (*`Extraction`* and *`Recursive Extraction`*) does not produce record projection constants anymore except for record projections explicitly instructed to extract, and records declared in opaque modules (#10577[262], fixes #7348[263], by Kazuhiko Sakaguchi).

Standard library

- Added `splitat` function and lemmas about `splitat` and `uncons` (#9379[264], by Yishuai Li, with help of Konstantinos Kallas, follow-up of #8365[265], which added `uncons` in 8.10+beta1).

[256] https://github.com/coq/coq/issues/9294
[257] https://github.com/coq/coq/pull/9918
[258] https://github.com/coq/coq/pull/10336
[259] https://github.com/coq/coq/issues/3219
[260] https://github.com/coq/coq/pull/10430
[261] https://github.com/coq/coq/issues/10361
[262] https://github.com/coq/coq/pull/10577
[263] https://github.com/coq/coq/issues/7348
[264] https://github.com/coq/coq/pull/9379
[265] https://github.com/coq/coq/pull/8365

3.2.5 Changes in 8.10.0

- Micromega tactics (*lia*, *nia*, etc) are no longer confused by primitive projections (#10806[266], fixes #9512[267] by Vincent Laporte).

3.2.6 Changes in 8.10.1

A few bug fixes and documentation improvements, in particular:

Kernel

- Fix proof of False when using `SProp` (incorrect De Bruijn handling when inferring the relevance mark of a function) (#10904[268], by Pierre-Marie Pédrot).

Tactics

- Fix an anomaly when unsolved evar in *Add Ring* (#10891[269], fixes #9851[270], by Gaëtan Gilbert).

Tactic language

- Fix Ltac regression in binding free names in uconstr (#10899[271], fixes #10894[272], by Hugo Herbelin).

CoqIDE

- Fix handling of unicode input before space (#10852[273], fixes #10842[274], by Arthur Charguéraud).

Extraction

- Fix custom extraction of inductives to JSON (#10897[275], fixes #4741[276], by Helge Bahmann).

3.2.7 Changes in 8.10.2

Kernel

- Fixed a critical bug of template polymorphism and nonlinear universes (#11128[277], fixes #11039[278], by Gaëtan Gilbert).
- Fixed an anomaly "Uncaught exception Constr.DestKO" on `Inductive` (#11052[279], fixes #11048[280], by Gaëtan Gilbert).
- Fixed an anomaly "not enough abstractions in fix body" (#11014[281], fixes #8459[282], by Gaëtan Gilbert).

Notations

[266] https://github.com/coq/coq/pull/10806
[267] https://github.com/coq/coq/issues/9512
[268] https://github.com/coq/coq/pull/10904
[269] https://github.com/coq/coq/pull/10891
[270] https://github.com/coq/coq/issues/9851
[271] https://github.com/coq/coq/pull/10899
[272] https://github.com/coq/coq/issues/10894
[273] https://github.com/coq/coq/pull/10852
[274] https://github.com/coq/coq/issues/10842
[275] https://github.com/coq/coq/pull/10897
[276] https://github.com/coq/coq/issues/4741
[277] https://github.com/coq/coq/pull/11128
[278] https://github.com/coq/coq/issues/11039
[279] https://github.com/coq/coq/pull/11052
[280] https://github.com/coq/coq/issues/11048
[281] https://github.com/coq/coq/pull/11014
[282] https://github.com/coq/coq/issues/8459

- Fixed an 8.10 regression related to the printing of coercions associated to notations (#11090[283], fixes #11033[284], by Hugo Herbelin).

CoqIDE

- Fixed uneven dimensions of CoqIDE panels when window has been resized (#11070[285], fixes 8.10-regression #10956[286], by Guillaume Melquiond).
- Do not include final stops in queries (#11069[287], fixes 8.10-regression #11058[288], by Guillaume Melquiond).

Infrastructure and dependencies

- Enable building of executables when they are running (#11000[289], fixes 8.9-regression #10728[290], by Gaëtan Gilbert).

3.3 Version 8.9

3.3.1 Summary of changes

Coq version 8.9 contains the result of refinements and stabilization of features and deprecations or removals of deprecated features, cleanups of the internals of the system and API along with a few new features. This release includes many user-visible changes, including deprecations that are documented in the next subsection and new features that are documented in the reference manual. Here are the most important changes:

- Kernel: mutually recursive records are now supported, by Pierre-Marie Pédrot.
- Notations:
  - Support for autonomous grammars of terms called "custom entries", by Hugo Herbelin (see Section *Custom entries* of the reference manual).
  - Deprecated notations of the standard library will be removed in the next version of Coq, see the next subsection for a script to ease porting, by Jason Gross and Jean-Christophe Léchenet.
  - Added the *`Numeral Notation`* command for registering decimal numeral notations for custom types, by Daniel de Rauglaudre, Pierre Letouzey and Jason Gross.
- Tactics: Introduction tactics *`intro`*/*`intros`* on a goal that is an existential variable now force a refinement of the goal into a dependent product rather than failing, by Hugo Herbelin.
- Decision procedures: deprecation of tactic `romega` in favor of *`lia`* and removal of `fourier`, replaced by *`lra`* which subsumes it, by Frédéric Besson, Maxime Dénès, Vincent Laporte and Laurent Théry.
- Proof language: focusing bracket `{` now supports named *goals*, e.g. `[x]:{` will focus on a goal (existential variable) named `x`, by Théo Zimmermann.
- SSReflect: the implementation of delayed clear was simplified by Enrico Tassi: the variables are always renamed using inaccessible names when the clear switch is processed and finally cleared at the end of the intro pattern. In addition to that, the use-and-discard flag `{}` typical of rewrite rules can now be also applied to views, e.g. `=> {}/v` applies `v` and then clears `v`. See Section *Introduction in the context*.

[283] https://github.com/coq/coq/pull/11090
[284] https://github.com/coq/coq/issues/11033
[285] https://github.com/coq/coq/pull/11070
[286] https://github.com/coq/coq/issues/10956
[287] https://github.com/coq/coq/pull/11069
[288] https://github.com/coq/coq/issues/11058
[289] https://github.com/coq/coq/pull/11000
[290] https://github.com/coq/coq/issues/10728

- Vernacular:
  - Experimental support for *attributes* on commands, by Vincent Laporte, as in `#[local] Lemma foo : bar.` Tactics and tactic notations now support the `deprecated` attribute.
  - Removed deprecated commands `Arguments Scope` and `Implicit Arguments` in favor of *`Arguments (scopes)`* and *`Arguments (implicits)`*, with the help of Jasper Hugunin.
  - New flag *`Uniform Inductive Parameters`* by Jasper Hugunin to avoid repeating uniform parameters in constructor declarations.
  - New commands *`Hint Variables`* and *`Hint Constants`*, by Matthieu Sozeau, for controlling the opacity status of variables and constants in hint databases. It is recommended to always use these commands after creating a hint database with *`Create HintDb`*.
  - Multiple sections with the same name are now allowed, by Jasper Hugunin.
- Library: additions and changes in the `VectorDef`, `Ascii`, and `String` libraries. Syntax notations are now available only when using `Import` of libraries and not merely `Require`, by various contributors (source of incompatibility, see the next subsection for details).
- Toplevels: `coqtop` and `coqide` can now display diffs between proof steps in color, using the *`Diffs`* option, by Jim Fehrle.
- Documentation: we integrated a large number of fixes to the new Sphinx documentation by various contributors, coordinated by Clément Pit-Claudel and Théo Zimmermann.
- Tools: removed the `gallina` utility and the homebrewed `Emacs` mode.
- Packaging: as in Coq 8.8.2, the Windows installer now includes many more external packages that can be individually selected for installation, by Michael Soegtrop.

Version 8.9 also comes with a bunch of smaller-scale changes and improvements regarding the different components of the system. Most important ones are documented in the next subsection file.

On the implementation side, the `dev/doc/changes.md` file documents the numerous changes to the implementation and improvements of interfaces. The file provides guidelines on porting a plugin to the new version and a plugin development tutorial kept in sync with Coq was introduced by Yves Bertot http://github.com/ybertot/plugin_tutorials. The new `dev/doc/critical-bugs` file documents the known critical bugs of Coq and affected releases.

The efficiency of the whole system has seen improvements thanks to contributions from Gaëtan Gilbert, Pierre-Marie Pédrot, and Maxime Dénès.

Maxime Dénès, Emilio Jesús Gallego Arias, Gaëtan Gilbert, Michael Soegtrop, Théo Zimmermann worked on maintaining and improving the continuous integration system.

The OPAM repository for Coq packages has been maintained by Guillaume Melquiond, Matthieu Sozeau, Enrico Tassi with contributions from many users. A list of packages is available at https://coq.inria.fr/opam/www/.

The 54 contributors for this version are Léo Andrès, Rin Arakaki, Benjamin Barenblat, Langston Barrett, Siddharth Bhat, Martin Bodin, Simon Boulier, Timothy Bourke, Joachim Breitner, Tej Chajed, Arthur Charguéraud, Pierre Courtieu, Maxime Dénès, Andres Erbsen, Jim Fehrle, Julien Forest, Emilio Jesus Gallego Arias, Gaëtan Gilbert, Matěj Grabovský, Jason Gross, Samuel Gruetter, Armaël Guéneau, Hugo Herbelin, Jasper Hugunin, Ralf Jung, Sam Pablo Kuper, Ambroise Lafont, Leonidas Lampropoulos, Vincent Laporte, Peter LeFanu Lumsdaine, Pierre Letouzey, Jean-Christophe Léchenet, Nick Lewycky, Yishuai Li, Sven M. Hallberg, Assia Mahboubi, Cyprien Mangin, Guillaume Melquiond, Perry E. Metzger, Clément Pit-Claudel, Pierre-Marie Pédrot, Daniel R. Grayson, Kazuhiko Sakaguchi, Michael Soegtrop, Matthieu Sozeau, Paul Steckler, Enrico Tassi, Laurent Théry, Anton Trunov, whitequark, Théo Winterhalter, Zeimer, Beta Ziliani, Théo Zimmermann.

Many power users helped to improve the design of the new features via the issue and pull request system, the Coq development mailing list or the coq-club@inria.fr mailing list. It would be impossible to mention exhaustively the names of everybody who to some extent influenced the development.

Version 8.9 is the fourth release of Coq developed on a time-based development cycle. Its development spanned 7 months from the release of Coq 8.8. The development moved to a decentralized merging process during this cycle. Guillaume Melquiond was in charge of the release process and is the maintainer of this release. This release is the result of ~2,000 commits and ~500 PRs merged, closing 75+ issues.

The Coq development team welcomed Vincent Laporte, a new Coq engineer working with Maxime Dénès in the Coq consortium.

Paris, November 2018,
Matthieu Sozeau for the Coq development team

3.3.2 Details of changes in 8.9+beta1

Kernel

- Mutually defined records are now supported.

Notations

- New support for autonomous grammars of terms, called ”custom entries” (see chapter ”Syntax extensions” of the reference manual).
- Deprecated compatibility notations will actually be removed in the next version of Coq. Uses of these notations are generally easy to fix thanks to the hint contained in the deprecation warnings. For projects that require more than a handful of such fixes, there is a script[291] that will do it automatically, using the output of `coqc`. The script contains documentation on its usage in a comment at the top.

Tactics

- Added toplevel goal selector `!` which expects a single focused goal. Use with `Set Default Goal Selector` to force focusing before tactics are called.
- The undocumented ”nameless” forms `fix N`, `cofix` that were deprecated in 8.8 have been removed from Ltac’s syntax; please use `fix ident N/cofix ident` to explicitly name the (co)fixpoint hypothesis to be introduced.
- Introduction tactics `intro`/`intros` on a goal that is an existential variable now force a refinement of the goal into a dependent product rather than failing.
- Support for `fix`/`cofix` added in Ltac `match` and `lazymatch`.
- Ltac backtraces now include trace information about tactics called by OCaml-defined tactics.
- Option `Ltac Debug` now applies also to terms built using Ltac functions.
- Deprecated the `Implicit Tactic` family of commands.
- The default program obligation tactic uses a bounded proof search instead of an unbounded and potentially non-terminating one now (source of incompatibility).
- The `simple apply` tactic now respects the `Opaque` flag when called from Ltac (`auto` still does not respect it).

[291] https://gist.github.com/JasonGross/9770653967de3679d131c59d42de6d17#file-replace-notations-py

- Tactic constr_eq now adds universe constraints needed for the identity to the context (it used to ignore them). New tactic constr_eq_strict checks that the required constraints already hold without adding new ones. Preexisting tactic constr_eq_nounivs can still be used if you really want to ignore universe constraints.
- Tactics and tactic notations now understand the deprecated attribute.
- The fourier tactic has been removed. Please now use lra instead. You may need to add Require Import Lra to your developments. For compatibility, we now define fourier as a deprecated alias of lra.
- The romega tactics have been deprecated; please use lia instead.

Focusing

- Focusing bracket { now supports named goal selectors, e.g. [x]: { will focus on a goal (existential variable) named x. As usual, unfocus with } once the sub-goal is fully solved.

Specification language

- A fix to unification (which was sensitive to the ascii name of variables) may occasionally change type inference in incompatible ways, especially regarding the inference of the return clause of match.

Standard Library

- Added Ascii.eqb and String.eqb and the =? notation for them, and proved some lemmas about them. Note that this might cause incompatibilities if you have, e.g., string_scope and Z_scope both open with string_scope on top, and expect =? to refer to Z.eqb. Solution: wrap _ =? _ in (_ =? _)%Z (or whichever scope you want).
- Added Ndigits.N2Bv_sized, and proved some lemmas about it. Deprecated Ndigits.N2Bv_gen.
- The scopes int_scope and uint_scope have been renamed to dec_int_scope and dec_uint_scope, to clash less with ssreflect and other packages. They are still delimited by %int and %uint.
- Syntax notations for string, ascii, Z, positive, N, R, and int31 are no longer available merely by *Require*ing the files that define the inductives. You must *Import* Coq.Strings.String.StringSyntax (after Require Coq.Strings.String), Coq.Strings.Ascii.AsciiSyntax (after Require Coq.Strings.Ascii), Coq.ZArith.BinIntDef, Coq.PArith.BinPosDef, Coq.NArith.BinNatDef, Coq.Reals.Rdefinitions, and Coq.Numbers.Cyclic.Int31.Int31, respectively, to be able to use these notations. Note that passing -compat 8.8 or issuing Require Import Coq.Compat.Coq88 will make these notations available. Users wishing to port their developments automatically may download fix.py from https://gist.github.com/JasonGross/5d4558edf8f5c2c548a3d96c17820169 and run a command like while true; do make -Okj 2>&1 | /path/to/fix.py; done and get a cup of coffee. (This command must be manually interrupted once the build finishes all the way though. Note also that this method is not fail-proof; you may have to adjust some scopes if you were relying on string notations not being available even when string_scope was open.)
- Numeral syntax for nat is no longer available without loading the entire prelude (Require Import Coq.Init.Prelude). This only impacts users running Coq without the init library (-nois or -noinit) and also issuing Require Import Coq.Init.Datatypes.

Tools

- Coq_makefile lets one override or extend the following variables from the command line: COQFLAGS, COQCHKFLAGS, COQDOCFLAGS. COQFLAGS is now entirely separate from COQLIBS, so in custom Makefiles $(COQFLAGS) should be replaced by $(COQFLAGS) $(COQLIBS).
- Removed the gallina utility (extracts specification from Coq vernacular files). If you would like to maintain this tool externally, please contact us.

- Removed the Emacs modes distributed with Coq. You are advised to use Proof-General[292] (and optionally Company-Coq[293]) instead. If your use case is not covered by these alternative Emacs modes, please open an issue. We can help set up external maintenance as part of Proof-General, or independently as part of coq-community.

Vernacular Commands

- Removed deprecated commands `Arguments Scope` and `Implicit Arguments` (not the option). Use the `Arguments` command instead.
- Nested proofs may be enabled through the option `Nested Proofs Allowed`. By default, they are disabled and produce an error. The deprecation warning which used to occur when using nested proofs has been removed.
- Added option `Uniform Inductive Parameters` which abstracts over parameters before typechecking constructors, allowing to write for example `Inductive list (A : Type) := nil : list | cons : A -> list -> list.`
- New `Set Hint Variables/Constants Opaque/Transparent` commands for setting globally the opacity flag of variables and constants in hint databases, overwriting the opacity set of the hint database.
- Added generic syntax for ”attributes”, as in: `#[local] Lemma foo : bar.`
- Added the `Numeral Notation` command for registering decimal numeral notations for custom types
- The `Set SsrHave NoTCResolution` command no longer has special global scope. If you want the previous behavior, use `Global Set SsrHave NoTCResolution`.
- Multiple sections with the same name are allowed.

Coq binaries and process model

- Before 8.9, Coq distributed a single `coqtop` binary and a set of dynamically loadable plugins that used to take over the main loop for tasks such as IDE language server or parallel proof checking.

  These plugins have been turned into full-fledged binaries so each different process has associated a particular binary now, in particular `coqidetop` is the CoqIDE language server, and `coq{proof,tactic,query}worker` are in charge of task-specific and parallel proof checking.

SSReflect

- The implementation of delayed clear switches in intro patterns is now simpler to explain:

  1. The immediate effect of a clear switch like `{x}` is to rename the variable `x` to `_x_` (i.e. a reserved identifier that cannot be mentioned explicitly)
  2. The delayed effect of `{x}` is that `_x_` is cleared at the end of the intro pattern
  3. A clear switch immediately before a view application like `{x}/v` is translated to `/v{x}`.

  In particular, the third rule lets one write `{x}/v` even if `v` uses the variable `x`: indeed the view is executed before the renaming.
- An empty clear switch is now accepted in intro patterns before a view application whenever the view is a variable. One can now write `{}/v` to mean `{v}/v`. Remark that `{}/x` is very similar to the idiom `{}e` for the rewrite tactic (the equation `e` is used for rewriting and then discarded).

Standard Library

- There are now conversions between `string` and `positive`, `Z`, `nat`, and `N` in binary, octal, and hex.

Display diffs between proof steps

[292] https://proofgeneral.github.io/

[293] https://github.com/cpitclaudel/company-coq

- `coqtop` and `coqide` can now highlight the differences between proof steps in color. This can be enabled from the command line or the `Set Diffs "on"/"off"/"removed"` command. Please see the documentation for details. Showing diffs in Proof General requires small changes to PG (under discussion).

Notations

- Added `++` infix for `VectorDef.append`. Note that this might cause incompatibilities if you have, e.g., `list_scope` and `vector_scope` both open with `vector_scope` on top, and expect `++` to refer to `app`. Solution: wrap `_ ++ _` in `(_ ++ _)%list` (or whichever scope you want).

3.3.3 Changes in 8.8.0

Various bug fixes.

3.3.4 Changes in 8.8.1

- Some quality-of-life fixes.
- Numerous improvements to the documentation.
- Fix a critical bug related to primitive projections and *`native_compute`*.
- Ship several additional Coq libraries with the Windows installer.

3.4 Version 8.8

3.4.1 Summary of changes

Coq version 8.8 contains the result of refinements and stabilization of features and deprecations, cleanups of the internals of the system along with a few new features. The main user visible changes are:

- Kernel: fix a subject reduction failure due to allowing fixpoints on non-recursive values, by Matthieu Sozeau. Handling of evars in the VM (the kernel still does not accept evars) by Pierre-Marie Pédrot.
- Notations: many improvements on recursive notations and support for destructuring patterns in the syntax of notations by Hugo Herbelin.
- Proof language: tacticals for profiling, timing and checking success or failure of tactics by Jason Gross. The focusing bracket `{` supports single-numbered goal selectors, e.g. `2:{`, by Théo Zimmermann.
- Vernacular: deprecation of commands and more uniform handling of the `Local` flag, by Vincent Laporte and Maxime Dénès, part of a larger attribute system overhaul. Experimental `Show Extraction` command by Pierre Letouzey. Coercion now accepts `Prop` or `Type` as a source by Arthur Charguéraud. `Export` modifier for options allowing to export the option to modules that `Import` and not only `Require` a module, by Pierre-Marie Pédrot.
- Universes: many user-level and API level enhancements: qualified naming and printing, variance annotations for cumulative inductive types, more general constraints and enhancements of the minimization heuristics, interaction with modules by Gaëtan Gilbert, Pierre-Marie Pédrot and Matthieu Sozeau.
- Library: Decimal Numbers library by Pierre Letouzey and various small improvements.
- Documentation: a large community effort resulted in the migration of the reference manual to the Sphinx documentation tool. The result is this manual. The new documentation infrastructure (based on Sphinx) is by Clément Pit-Claudel. The migration was coordinated by Maxime Dénès and Paul

Steckler, with some help of Théo Zimmermann during the final integration phase. The 14 people who ported the manual are Calvin Beck, Heiko Becker, Yves Bertot, Maxime Dénès, Richard Ford, Pierre Letouzey, Assia Mahboubi, Clément Pit-Claudel, Laurence Rideau, Matthieu Sozeau, Paul Steckler, Enrico Tassi, Laurent Théry, Nikita Zyuzin.

- Tools: experimental `-mangle-names` option to `coqtop/coqc` for linting proof scripts, by Jasper Hugunin.

On the implementation side, the `dev/doc/changes.md` file documents the numerous changes to the implementation and improvements of interfaces. The file provides guidelines on porting a plugin to the new version.

Version 8.8 also comes with a bunch of smaller-scale changes and improvements regarding the different components of the system. Most important ones are documented in the next subsection file.

The efficiency of the whole system has seen improvements thanks to contributions from Gaëtan Gilbert, Pierre-Marie Pédrot, Maxime Dénès and Matthieu Sozeau and performance issue tracking by Jason Gross and Paul Steckler.

The official wiki and the bugtracker of Coq migrated to the GitHub platform, thanks to the work of Pierre Letouzey and Théo Zimmermann. Gaëtan Gilbert, Emilio Jesús Gallego Arias worked on maintaining and improving the continuous integration system.

The OPAM repository for Coq packages has been maintained by Guillaume Melquiond, Matthieu Sozeau, Enrico Tassi with contributions from many users. A list of packages is available at https://coq.inria.fr/opam/www/.

The 44 contributors for this version are Yves Bertot, Joachim Breitner, Tej Chajed, Arthur Charguéraud, Jacques-Pascal Deplaix, Maxime Dénès, Jim Fehrle, Julien Forest, Yannick Forster, Gaëtan Gilbert, Jason Gross, Samuel Gruetter, Thomas Hebb, Hugo Herbelin, Jasper Hugunin, Emilio Jesus Gallego Arias, Ralf Jung, Johannes Kloos, Matej Košík, Robbert Krebbers, Tony Beta Lambda, Vincent Laporte, Peter LeFanu Lumsdaine, Pierre Letouzey, Farzon Lotfi, Cyprien Mangin, Guillaume Melquiond, Raphaël Monat, Carl Patenaude Poulin, Pierre-Marie Pédrot, Clément Pit-Claudel, Matthew Ryan, Matt Quinn, Sigurd Schneider, Bernhard Schommer, Michael Soegtrop, Matthieu Sozeau, Arnaud Spiwack, Paul Steckler, Enrico Tassi, Anton Trunov, Martin Vassor, Vadim Zaliva and Théo Zimmermann.

Version 8.8 is the third release of Coq developed on a time-based development cycle. Its development spanned 6 months from the release of Coq 8.7 and was based on a public roadmap. The development process was coordinated by Matthieu Sozeau. Maxime Dénès was in charge of the release process. Théo Zimmermann is the maintainer of this release.

Many power users helped to improve the design of the new features via the bug tracker, the pull request system, the Coq development mailing list or the coq-club@inria.fr mailing list. Special thanks to the users who contributed patches and intensive brain-storming and code reviews, starting with Jason Gross, Ralf Jung, Robbert Krebbers and Amin Timany. It would however be impossible to mention exhaustively the names of everybody who to some extent influenced the development.

The Coq consortium, an organization directed towards users and supporters of the system, is now running and employs Maxime Dénès. The contacts of the Coq Consortium are Yves Bertot and Maxime Dénès.

Santiago de Chile, March 2018,
Matthieu Sozeau for the Coq development team

3.4.2 Details of changes in 8.8+beta1

Kernel

- Support for template polymorphism for definitions was removed. May trigger more ”universe inconsistency” errors in rare occasions.
- Fixpoints are no longer allowed on non-recursive inductive types.

Notations

- Recursive notations with the recursive pattern repeating on the right (e.g. ”(x ; .. ; y ; z)”) now supported.
- Notations with a specific level for the leftmost nonterminal, when printing-only, are supported.
- Notations can now refer to the syntactic category of patterns (as in ”fun 'pat =>” or ”match p with pat => ... end”). Two variants are available, depending on whether a single variable is considered as a pattern or not.
- Recursive notations now support ”..” patterns with several occurrences of the recursive term or binder, possibly mixing terms and binders, possibly in reverse left-to-right order.
- ”Locate” now working also on notations of the form ”x + y” (rather than ”_ + _”).

Specification language

- When printing clauses of a ”match”, clauses with same right-hand side are factorized and the last most factorized clause with no variables, if it exists, is turned into a default clause. Use ”Unset Printing Allow Default Clause” do deactivate printing of a default clause. Use ”Unset Printing Factorizable Match Patterns” to deactivate factorization of clauses with same right-hand side.

Tactics

- On Linux, ”native_compute” calls can be profiled using the ”perf” utility. The command ”Set NativeCompute Profiling” enables profiling, and ”Set NativeCompute Profile Filename” customizes the profile filename.
- The tactic ”omega” is now aware of the bodies of context variables such as ”x := 5 : Z” (see #1362). This could be disabled via Unset Omega UseLocalDefs.
- The tactic ”romega” is also aware now of the bodies of context variables.
- The tactic ”zify” resp. ”omega with N” is now aware of N.pred.
- Tactic ”decide equality” now able to manage constructors which contain proofs.
- Added tactics reset ltac profile, show ltac profile (and variants)
- Added tactics restart_timer, finish_timing, and time_constr as an experimental way of timing Ltac's evaluation phase
- Added tactic optimize_heap, analogous to the Vernacular Optimize Heap, which performs a major garbage collection and heap compaction in the OCaml run-time system.
- The tactics ”dtauto”, ”dintuition”, ”firstorder” now handle inductive types with let bindings in the parameters.
- The tactic `dtauto` now handles some inductives such as `@sigT A (fun _ => B)` as non-dependent conjunctions.
- A bug fixed in `rewrite H in *` and `rewrite H in * |-` may cause a few rare incompatibilities (it was unintendedly recursively rewriting in the side conditions generated by H).

- Added tactics ”assert_succeeds tac” and ”assert_fails tac” to ensure properties of the execution of a tactic without keeping the effect of the execution.
- `vm_compute` now supports existential variables.
- Calls to `shelve` and `give_up` within calls to tactic `refine` now working.
- Deprecated tactic `appcontext` was removed.

Focusing

- Focusing bracket `{` now supports single-numbered goal selector, e.g. `2: {` will focus on the second subgoal. As usual, unfocus with `}` once the sub-goal is fully solved. The `Focus` and `Unfocus` commands are now deprecated.

Vernacular Commands

- Proofs ending in ”Qed exporting ident, .., ident” are not supported anymore. Constants generated during `abstract` are kept private to the local environment.
- The deprecated Coercion Local, Open Local Scope, Notation Local syntax was removed. Use Local as a prefix instead.
- For the Extraction Language command, ”OCaml” is spelled correctly. The older ”Ocaml” is still accepted, but deprecated.
- Using “Require” inside a section is deprecated.
- An experimental command ”Show Extraction” allows to extract the content of the current ongoing proof (grant wish #4129).
- Coercion now accepts the type of its argument to be ”Prop” or ”Type”.
- The ”Export” modifier can now be used when setting and unsetting options, and will result in performing the same change when the module corresponding the command is imported.
- The `Axiom` command does not automatically declare axioms as instances when their type is a class. Previous behavior can be restored using `Set Typeclasses Axioms Are Instances`.

Universes

- Qualified naming of global universes now works like other namespaced objects (e.g. constants), with a separate namespace, inside and across module and library boundaries. Global universe names introduced in an inductive / constant / Let declaration get qualified with the name of the declaration.
- Universe cumulativity for inductive types is now specified as a variance for each polymorphic universe. See the reference manual for more information.
- Inference of universe constraints with cumulative inductive types produces more general constraints. Unsetting new option Cumulativity Weak Constraints produces even more general constraints (but may produce too many universes to be practical).
- Fix #5726: Notations that start with `Type` now support universe instances with `@{u}`.
- `with Definition` now understands universe declarations (like `@{u| Set < u}`).

Tools

- Coq can now be run with the option -mangle-names to change the auto-generated name scheme. This is intended to function as a linter for developments that want to be robust to changes in auto-generated names. This feature is experimental, and may change or disappear without warning.
- GeoProof support was removed.

Checker

- The checker now accepts filenames in addition to logical paths.

CoqIDE

- Find and Replace All report the number of occurrences found; Find indicates when it wraps.

coqdep

- Learned to read -I, -Q, -R and filenames from _CoqProject files. This is used by coq_makefile when generating dependencies for .v files (but not other files).

Documentation

- The Coq FAQ, formerly located at https://coq.inria.fr/faq, has been moved to the GitHub wiki section of this repository; the main entry page is https://github.com/coq/coq/wiki/The-Coq-FAQ.
- Documentation: a large community effort resulted in the migration of the reference manual to the Sphinx documentation tool. The result is partially integrated in this version.

Standard Library

- New libraries Coq.Init.Decimal, Coq.Numbers.DecimalFacts, Coq.Numbers.DecimalNat, Coq.Numbers.DecimalPos, Coq.Numbers.DecimalN, Coq.Numbers.DecimalZ, Coq.Numbers.DecimalString providing a type of decimal numbers, some facts about them, and conversions between decimal numbers and nat, positive, N, Z, and string.
- Added [Coq.Strings.String.concat] to concatenate a list of strings inserting a separator between each item
- Notation `'` for Zpos in QArith was removed.
- Some deprecated aliases are now emitting warnings when used.

Compatibility support

- Support for compatibility with versions before 8.6 was dropped.

Options

- The following deprecated options have been removed:
  - `Refolding Reduction`
  - `Standard Proposition Elimination`
  - `Dependent Propositions Elimination`
  - `Discriminate Introduction`
  - `Shrink Abstract`
  - `Tactic Pattern Unification`
  - `Intuition Iff Unfolding`
  - `Injection L2R Pattern Order`
  - `Record Elimination Schemes`
  - `Match Strict`
  - `Tactic Compat Context`
  - `Typeclasses Legacy Resolution`
  - `Typeclasses Module Eta`
  - `Typeclass Resolution After Apply`

3.4.3 Details of changes in 8.8.0

Tools

- Asynchronous proof delegation policy was fixed. Since version 8.7 Coq was ignoring previous runs and the `-async-proofs-delegation-threshold` option did not have the expected behavior.

Tactic language

- The undocumented ”nameless” forms `fix N`, `cofix` have been deprecated; please use `fix ident N /cofix ident` to explicitly name the (co)fixpoint hypothesis to be introduced.

Documentation

- The reference manual is now fully ported to Sphinx.

Other small deprecations and bug fixes.

3.4.4 Details of changes in 8.8.1

Kernel

- Fix a critical bug with cofixpoints and `vm_compute/native_compute` (#7333).
- Fix a critical bug with modules and algebraic universes (#7695)
- Fix a critical bug with inlining of polymorphic constants (#7615).
- Fix a critical bug with universe polymorphism and `vm_compute` (#7723). Was present since 8.5.

Notations

- Fixed unexpected collision between only-parsing and only-printing notations (issue #7462).

Windows installer

- The Windows installer now includes external packages Ltac2 and Equations (it included the Bignums package since 8.8+beta1).

Many other bug fixes, documentation improvements (including fixes of regressions due to the Sphinx migration), and user message improvements (for details, see the 8.8.1 milestone at https://github.com/coq/coq/milestone/13?closed=1).

3.4.5 Details of changes in 8.8.2

Documentation

- A PDF version of the reference manual is available once again.

Tools

- The coq-makefile targets `print-pretty-timed`, `print-pretty-timed-diff`, and `print-pretty-single-time-diff` now correctly label the ”before” and ”after” columns, rather than swapping them.

Kernel

- The kernel does not tolerate capture of global universes by polymorphic universe binders, fixing a soundness break (triggered only through custom plugins)

Windows installer

- The Windows installer now includes many more external packages that can be individually selected for installation.

Many other bug fixes and lots of documentation improvements (for details, see the 8.8.2 milestone at https://github.com/coq/coq/milestone/15?closed=1).

3.5 Version 8.7

3.5.1 Summary of changes

Coq version 8.7 contains the result of refinements, stabilization of features and cleanups of the internals of the system along with a few new features. The main user visible changes are:

- New tactics: variants of tactics supporting existential variables *eassert*, *eenough*, etc... by Hugo Herbelin. Tactics `extensionality in H` and *inversion_sigma* by Jason Gross, `specialize with ...` accepting partial bindings by Pierre Courtieu.
- `Cumulative Polymorphic Inductive` types, allowing cumulativity of universes to go through applied inductive types, by Amin Timany and Matthieu Sozeau.
- Integration of the SSReflect plugin and its documentation in the reference manual, by Enrico Tassi, Assia Mahboubi and Maxime Dénès.
- The `coq_makefile` tool was completely redesigned to improve its maintainability and the extensibility of generated Makefiles, and to make `_CoqProject` files more palatable to IDEs by Enrico Tassi.

Coq 8.7 involved a large amount of work on cleaning and speeding up the code base, notably the work of Pierre-Marie Pédrot on making the tactic-level system insensitive to existential variable expansion, providing a safer API to plugin writers and making the code more robust. The `dev/doc/changes.txt` file documents the numerous changes to the implementation and improvements of interfaces. An effort to provide an official, streamlined API to plugin writers is in progress, thanks to the work of Matej Košík.

Version 8.7 also comes with a bunch of smaller-scale changes and improvements regarding the different components of the system. We shall only list a few of them.

The efficiency of the whole system has been significantly improved thanks to contributions from Pierre-Marie Pédrot, Maxime Dénès and Matthieu Sozeau and performance issue tracking by Jason Gross and Paul Steckler.

Thomas Sibut-Pinote and Hugo Herbelin added support for side effect hooks in cbv, cbn and simpl. The side effects are provided via a plugin available at https://github.com/herbelin/reduction-effects/.

The BigN, BigZ, BigQ libraries are no longer part of the Coq standard library, they are now provided by a separate repository https://github.com/coq/bignums, maintained by Pierre Letouzey.

In the Reals library, `IZR` has been changed to produce a compact representation of integers and real constants are now represented using `IZR` (work by Guillaume Melquiond).

Standard library additions and improvements by Jason Gross, Pierre Letouzey and others, documented in the next subsection file.

The mathematical proof language/declarative mode plugin was removed from the archive.

The OPAM repository for Coq packages has been maintained by Guillaume Melquiond, Matthieu Sozeau, Enrico Tassi with contributions from many users. A list of packages is available at https://coq.inria.fr/opam/www/.

Packaging tools and software development kits were prepared by Michael Soegtrop with the help of Maxime Dénès and Enrico Tassi for Windows, and Maxime Dénès for MacOS X. Packages are regularly built on the Travis continuous integration server.

The contributors for this version are Abhishek Anand, C.J. Bell, Yves Bertot, Frédéric Besson, Tej Chajed, Pierre Courtieu, Maxime Dénès, Julien Forest, Gaëtan Gilbert, Jason Gross, Hugo Herbelin, Emilio Jesús Gallego Arias, Ralf Jung, Matej Košík, Xavier Leroy, Pierre Letouzey, Assia Mahboubi, Cyprien Mangin, Erik Martin-Dorel, Olivier Marty, Guillaume Melquiond, Sam Pablo Kuper, Benjamin Pierce, Pierre-Marie Pédrot, Lars Rasmusson, Lionel Rieg, Valentin Robert, Yann Régis-Gianas, Thomas Sibut-Pinote, Michael Soegtrop, Matthieu Sozeau, Arnaud Spiwack, Paul Steckler, George Stelle, Pierre-Yves Strub, Enrico Tassi, Hendrik Tews, Amin Timany, Laurent Théry, Vadim Zaliva and Théo Zimmermann.

The development process was coordinated by Matthieu Sozeau with the help of Maxime Dénès, who was also in charge of the release process. Théo Zimmermann is the maintainer of this release.

Many power users helped to improve the design of the new features via the bug tracker, the pull request system, the Coq development mailing list or the Coq-Club mailing list. Special thanks to the users who contributed patches and intensive brain-storming and code reviews, starting with Jason Gross, Ralf Jung, Robbert Krebbers, Xavier Leroy, Clément Pit–Claudel and Gabriel Scherer. It would however be impossible to mention exhaustively the names of everybody who to some extent influenced the development.

Version 8.7 is the second release of Coq developed on a time-based development cycle. Its development spanned 9 months from the release of Coq 8.6 and was based on a public road-map. It attracted many external contributions. Code reviews and continuous integration testing were systematically used before integration of new features, with an important focus given to compatibility and performance issues, resulting in a hopefully more robust release than Coq 8.6 while maintaining compatibility.

Coq Enhancement Proposals (CEPs for short) and open pull request discussions were used to discuss publicly the new features.

The Coq consortium, an organization directed towards users and supporters of the system, is now upcoming and will rely on Inria's newly created Foundation.

Paris, August 2017,
Matthieu Sozeau and the Coq development team

3.5.2 Potential compatibility issues

- Extra superfluous names in introduction patterns may now raise an error rather than a warning when the superfluous name is already in use. The easy fix is to remove the superfluous name.

3.5.3 Details of changes in 8.7+beta1

Tactics

- New tactic "extensionality in H" which applies (possibly dependent) functional extensionality in H supposed to be a quantified equality until giving a bare equality.
- New tactic `inversion_sigma` which turns equalities of dependent pairs (e.g., `existT P x p = existT P y q`, frequently left over by `inversion` on a dependent type family) into pairs of equalities (e.g., a hypothesis `H : x = y` and a hypothesis of type `rew H in p = q`); these hypotheses can subsequently be simplified using `subst`, without ever invoking any kind of axiom asserting uniqueness of identity proofs. If you want to explicitly specify the hypothesis to be inverted, or name the generated hypotheses,

you can invoke `induction H as [H1 H2] using eq_sigT_rect`. The tactic also works for `sig`, `sigT2`, and `sig2`, and there are similar `eq_sig*_rect` induction lemmas.

- Tactic ”specialize with ...” now accepts any partial bindings. Missing bindings are either solved by unification or left quantified in the hypothesis.
- New representation of terms that statically ensure stability by evar-expansion. This has several consequences.
  - In terms of performance, this adds a cost to every term destructuration, but at the same time most eager evar normalizations were removed, which couterbalances this drawback and even sometimes outperforms the old implementation. For instance, many operations that would require O(n) normalization of the term are now O(1) in tactics. YMMV.
  - This triggers small changes in unification, which was not evar-insensitive. Most notably, the new implementation recognizes Miller patterns that were missed before because of a missing normalization step. Hopefully this should be fairly uncommon.
- Tactic ”auto with real” can now discharge comparisons of literals.
- The types of variables in patterns of ”match” are now beta-iota-reduced after type-checking. This has an impact on the type of the variables that the tactic ”refine” introduces in the context, producing types a priori closer to the expectations.
- In ”Tactic Notation” or ”TACTIC EXTEND”, entry ”constr_with_bindings” now uses type classes and rejects terms with unresolved holes, like entry ”constr” does. To get the former behavior use ”open_constr_with_bindings” (possible source of incompatibility).
- New e-variants eassert, eenough, epose proof, eset, eremember, epose which behave like the corresponding variants with no ”e” but turn unresolved implicit arguments into existential variables, on the shelf, rather than failing.
- Tactic injection has become more powerful (closes bug #4890) and its documentation has been updated.
- New variants of the `first` and `solve` tacticals that do not rely on parsing rules, meant to define tactic notations.
- Added support for side effects hooks in `cbv`, `cbn` and `simpl`. The side effects are provided via a plugin: https://github.com/herbelin/reduction-effects/
- It is now possible to take hint database names as parameters in a Ltac definition or a Tactic Notation.
- New option `Set Ltac Batch Debug` on top of `Set Ltac Debug` for non-interactive Ltac debug output.

Gallina

- Now supporting all kinds of binders, including ’pat, in syntax of record fields.

Vernacular Commands

- Goals context can be printed in a more compact way when `Set Printing Compact Contexts` is activated.
- Unfocused goals can be printed with the `Set Printing Unfocused` option.
- `Print` now shows the types of let-bindings.
- The compatibility options for printing primitive projections (`Set Printing Primitive Projection Parameters` and `Set Printing Primitive Projection Compatibility`) are now off by default.
- Possibility to unset the printing of notations in a more fine grained fashion than `Unset Printing Notations` is provided without any user-syntax. The goal is that someone creates a plugin to experiment such a user-syntax, to be later integrated in Coq when stabilized.
- `About` now tells if a reference is a coercion.

- The deprecated `Save` vernacular and its form `Save Theorem id` to close proofs have been removed from the syntax. Please use `Qed`.
- `Search` now sorts results by relevance (the relevance metric is a weighted sum of number of distinct symbols and size of the term).

Standard Library

- New file PropExtensionality.v to explicitly work in the axiomatic context of propositional extensionality.
- New file SetoidChoice.v axiomatically providing choice over setoids, and, consequently, choice of representatives in equivalence classes. Various proof-theoretic characterizations of choice over setoids in file ChoiceFacts.v.
- New lemmas about iff and about orders on positive and Z.
- New lemmas on powerRZ.
- Strengthened statement of JMeq_eq_dep (closes bug #4912).
- The BigN, BigZ, BigZ libraries are no longer part of the Coq standard library, they are now provided by a separate repository https://github.com/coq/bignums The split has been done just after the Int31 library.
- IZR (Reals) has been changed to produce a compact representation of integers. As a consequence, IZR is no longer convertible to INR and lemmas such as INR_IZR_INZ should be used instead.
- Real constants are now represented using IZR rather than R0 and R1; this might cause rewriting rules to fail to apply to constants.
- Added new notation {x & P} for sigT (without a type for x)

Plugins

- The Ssreflect plugin is now distributed with Coq. Its documentation has been integrated as a chapter of the reference manual. This chapter is work in progress so feedback is welcome.
- The mathematical proof language (also known as declarative mode) was removed.
- A new command Extraction TestCompile has been introduced, not meant for the general user but instead for Coq's test-suite.
- The extraction plugin is no longer loaded by default. It must be explicitly loaded with [Require Extraction], which is backwards compatible.
- The functional induction plugin (which provides the [Function] vernacular) is no longer loaded by default. It must be explicitly loaded with [Require FunInd], which is backwards compatible.

Dependencies

- Support for camlp4 has been removed.

Tools

- coq_makefile was completely redesigned to improve its maintainability and the extensibility of generated Makefiles, and to make _CoqProject files more palatable to IDEs. Overview:
  - _CoqProject files contain only Coq specific data (i.e. the list of files, -R options, ...)
  - coq_makefile translates _CoqProject to Makefile.conf and copies in the desired location a standard Makefile (that reads Makefile.conf)
  - Makefile extensions can be implemented in a Makefile.local file (read by the main Makefile) by installing a hook in the extension points provided by the standard Makefile

The current version contains code for retro compatibility that prints warnings when a deprecated feature is used. Please upgrade your _CoqProject accordingly.

  - Additionally, coq_makefile-made Makefiles now support experimental timing targets `pretty-timed`, `pretty-timed-before`, `pretty-timed-after`, `print-pretty-timed-diff`, `print-pretty-single-time-diff`, `all.timing.diff`, and the variable `TIMING=1` (or `TIMING=before` or `TIMING=after`); see the documentation for more details.

Build Infrastructure

- Note that 'make world' does not build the bytecode binaries anymore. For that, you can use 'make byte' (and 'make install-byte' afterwards). Warning: native and byte compilations should *not* be mixed in the same instance of 'make -j', otherwise both ocamlc and ocamlopt might race for access to the same .cmi files. In short, use "make -j && make -j byte" instead of "make -j world byte".

Universes

- Cumulative inductive types. see prefixes "Cumulative", "NonCumulative" for inductive definitions and the option "Set Polymorphic Inductive Cumulativity" in the reference manual.
- New syntax `foo@{_}` to instantiate a polymorphic definition with anonymous universes (can also be used with `Type`).

XML Protocol and internal changes

See dev/doc/changes.txt

Many bugfixes including #1859, #2884, #3613, #3943, #3994, #4250, #4709, #4720, #4824, #4844, #4911, #5026, #5233, #5275, #5315, #5336, #5360, #5390, #5414, #5417, #5420, #5439, #5449, #5475, #5476, #5482, #5501, #5507, #5520, #5523, #5524, #5553, #5577, #5578, #5589, #5597, #5598, #5607, #5618, #5619, #5620, #5641, #5648, #5651, #5671.

Many bugfixes on OS X and Windows (now the test-suite passes on these platforms too).

Many optimizations.

Many documentation improvements.

3.5.4 Details of changes in 8.7+beta2

Tools

- In CoqIDE, the "Compile Buffer" command takes account of flags in _CoqProject or other project file.

Improvements around some error messages.

Many bug fixes including two important ones:

- Bug #5730: CoqIDE becomes unresponsive on file open.
- coq_makefile: make sure compile flags for Coq and coq_makefile are in sync (in particular, make sure the `-safe-string` option is used to compile plugins).

3.5.5 Details of changes in 8.7.0

OCaml

- Users can pass specific flags to the OCaml optimizing compiler by -using the flambda-opts configure-time option.

  Beware that compiling Coq with a flambda-enabled compiler is experimental and may require large amounts of RAM and CPU, see INSTALL for more details.

3.5.6 Details of changes in 8.7.1

Compatibility with OCaml 4.06.0.

Many bug fixes, documentation improvements, and user message improvements (for details see the 8.7.1 milestone at https://github.com/coq/coq/milestone/10?closed=1).

3.5.7 Details of changes in 8.7.2

Fixed a critical bug in the VM handling of universes (#6677). This bug affected all releases since 8.5.

Improved support for building with OCaml 4.06.0 and external num package.

Many other bug fixes, documentation improvements, and user message improvements (for details, see the 8.7.2 milestone at https://github.com/coq/coq/milestone/11?closed=1).

3.6 Version 8.6

3.6.1 Summary of changes

Coq version 8.6 contains the result of refinements, stabilization of 8.5's features and cleanups of the internals of the system. Over the year of (now time-based) development, about 450 bugs were resolved and over 100 contributions integrated. The main user visible changes are:

- A new, faster state-of-the-art universe constraint checker, by Jacques-Henri Jourdan.
- In CoqIDE and other asynchronous interfaces, more fine-grained asynchronous processing and error reporting by Enrico Tassi, making Coq capable of recovering from errors and continue processing the document.
- More access to the proof engine features from Ltac: goal management primitives, range selectors and a *typeclasses eauto* engine handling multiple goals and multiple successes, by Cyprien Mangin, Matthieu Sozeau and Arnaud Spiwack.
- Tactic behavior uniformization and specification, generalization of intro-patterns by Hugo Herbelin and others.
- A brand new warning system allowing to control warnings, turn them into errors or ignore them selectively by Maxime Dénès, Guillaume Melquiond, Pierre-Marie Pédrot and others.
- Irrefutable patterns in abstractions, by Daniel de Rauglaudre.
- The ssreflect subterm selection algorithm by Georges Gonthier and Enrico Tassi is now accessible to tactic writers through the ssrmatching plugin.
- Integration of LtacProf, a profiler for Ltac by Jason Gross, Paul Steckler, Enrico Tassi and Tobias Tebbi.

Coq 8.6 also comes with a bunch of smaller-scale changes and improvements regarding the different components of the system. We shall only list a few of them.

The iota reduction flag is now a shorthand for match, fix and cofix flags controlling the corresponding reduction rules (by Hugo Herbelin and Maxime Dénès).

Maxime Dénès maintained the native compilation machinery.

Pierre-Marie Pédrot separated the Ltac code from general purpose tactics, and generalized and rationalized the handling of generic arguments, allowing to create new versions of Ltac more easily in the future.

In patterns and terms, @, abbreviations and notations are now interpreted the same way, by Hugo Herbelin.

Name handling for universes has been improved by Pierre-Marie Pédrot and Matthieu Sozeau. The minimization algorithm has been improved by Matthieu Sozeau.

The unifier has been improved by Hugo Herbelin and Matthieu Sozeau, fixing some incompatibilities introduced in Coq 8.5. Unification constraints can now be left floating around and be seen by the user thanks to a new option. The Keyed Unification mode has been improved by Matthieu Sozeau.

The typeclass resolution engine and associated proof-search tactic have been reimplemented on top of the proof-engine monad, providing better integration in tactics, and new options have been introduced to control it, by Matthieu Sozeau with help from Théo Zimmermann.

The efficiency of the whole system has been significantly improved thanks to contributions from Pierre-Marie Pédrot, Maxime Dénès and Matthieu Sozeau and performance issue tracking by Jason Gross and Paul Steckler.

Standard library improvements by Jason Gross, Sébastien Hinderer, Pierre Letouzey and others.

Emilio Jesús Gallego Arias contributed many cleanups and refactorings of the pretty-printing and user interface communication components.

Frédéric Besson maintained the micromega tactic.

The OPAM repository for Coq packages has been maintained by Guillaume Claret, Guillaume Melquiond, Matthieu Sozeau, Enrico Tassi and others. A list of packages is now available at https://coq.inria.fr/opam/www/.

Packaging tools and software development kits were prepared by Michael Soegtrop with the help of Maxime Dénès and Enrico Tassi for Windows, and Maxime Dénès and Matthieu Sozeau for MacOS X. Packages are now regularly built on the continuous integration server. Coq now comes with a META file usable with ocamlfind, contributed by Emilio Jesús Gallego Arias, Gregory Malecha, and Matthieu Sozeau.

Matej Košík maintained and greatly improved the continuous integration setup and the testing of Coq contributions. He also contributed many API improvements and code cleanups throughout the system.

The contributors for this version are Bruno Barras, C.J. Bell, Yves Bertot, Frédéric Besson, Pierre Boutillier, Tej Chajed, Guillaume Claret, Xavier Clerc, Pierre Corbineau, Pierre Courtieu, Maxime Dénès, Ricky Elrod, Emilio Jesús Gallego Arias, Jason Gross, Hugo Herbelin, Sébastien Hinderer, Jacques-Henri Jourdan, Matej Košík, Xavier Leroy, Pierre Letouzey, Gregory Malecha, Cyprien Mangin, Erik Martin-Dorel, Guillaume Melquiond, Clément Pit–Claudel, Pierre-Marie Pédrot, Daniel de Rauglaudre, Lionel Rieg, Gabriel Scherer, Thomas Sibut-Pinote, Matthieu Sozeau, Arnaud Spiwack, Paul Steckler, Enrico Tassi, Laurent Théry, Nickolai Zeldovich and Théo Zimmermann. The development process was coordinated by Hugo Herbelin and Matthieu Sozeau with the help of Maxime Dénès, who was also in charge of the release process.

Many power users helped to improve the design of the new features via the bug tracker, the pull request system, the Coq development mailing list or the Coq-Club mailing list. Special thanks to the users who contributed patches and intensive brain-storming and code reviews, starting with Cyril Cohen, Jason Gross, Robbert Krebbers, Jonathan Leivent, Xavier Leroy, Gregory Malecha, Clément Pit–Claudel, Gabriel Scherer and Beta Ziliani. It would however be impossible to mention exhaustively the names of everybody who to some extent influenced the development.

Version 8.6 is the first release of Coq developed on a time-based development cycle. Its development spanned 10 months from the release of Coq 8.5 and was based on a public roadmap. To date, it contains more external contributions than any previous Coq system. Code reviews were systematically done before integration of new features, with an important focus given to compatibility and performance issues, resulting in a hopefully more robust release than Coq 8.5.

Coq Enhancement Proposals (CEPs for short) were introduced by Enrico Tassi to provide more visibility and a discussion period on new features, they are publicly available https://github.com/coq/ceps.

Started during this period, an effort is led by Yves Bertot and Maxime Dénès to put together a Coq consortium.

Paris, November 2016,
Matthieu Sozeau and the Coq development team

3.6.2 Potential sources of incompatibilities

- Symptom: An obligation generated by Program or an abstracted subproof has different arguments.

  Cause: Set Shrink Abstract and Set Shrink Obligations are on by default and the subproof does not use the argument.

  Remedy:

  - Adapt the script.
  - Write an explicit lemma to prove the obligation/subproof and use it instead (compatible with 8.4).
  - Unset the option for the program/proof the obligation/subproof originates from.

- Symptom: In a goal, order of hypotheses, or absence of an equality of the form ”x = t” or ”t = x”, or no unfolding of a local definition.

  Cause: This might be connected to a number of fixes in the tactic ”subst”. The former behavior can be reactivated by issuing ”Unset Regular Subst Tactic”.

3.6.3 Details of changes in 8.6beta1

Kernel

- A new, faster state-of-the-art universe constraint checker.

Specification language

- Giving implicit arguments explicitly to a constant with multiple choices of implicit arguments does not break any more insertion of further maximal implicit arguments.
- Ability to put any pattern in binders, prefixed by quote, e.g. ”fun ’(a,b) => ...”, ”λ ’(a,(b,c)), ...”, ”Definition foo ’(x,y) := ...”. It expands into a ”let ’pattern := ...”

Tactics

- Flag ”Bracketing Last Introduction Pattern” is now on by default.
- Flag ”Regular Subst Tactic” is now on by default: it respects the initial order of hypothesis, it contracts cycles, it unfolds no local definitions (common source of incompatibilities, fixable by ”Unset Regular Subst Tactic”).
- New flag ”Refolding Reduction”, now disabled by default, which turns on refolding of constants/fixpoints (as in cbn) during the reductions done during type inference and tactic retyping. Can be extremely expensive. When set off, this recovers the 8.4 behaviour of unification and type inference. Potential source of incompatibility with 8.5 developments (the option is set on in Compat/Coq85.v).
- New flag ”Shrink Abstract” that minimalizes proofs generated by the abstract tactical w.r.t. variables appearing in the body of the proof. On by default and deprecated. Minor source of incompatibility for code relying on the precise arguments of abstracted proofs.

- Serious bugs are fixed in tactic ”double induction” (source of incompatibilities as soon as the inductive types have dependencies in the type of their constructors; ”double induction” remains however deprecated).
- In introduction patterns of the form (pat1,...,patn), n should match the exact number of hypotheses introduced (except for local definitions for which pattern can be omitted, as in regular pattern-matching).
- Tactic scopes in Ltac like constr: and ltac: now require parentheses around their argument.
- Every generic argument type declares a tactic scope of the form ”name:(...)” where name is the name of the argument. This generalizes the constr: and ltac: instances.
- When in strict mode (i.e. in a Ltac definition), if the ”intro” tactic is given a free identifier, it is not bound in subsequent tactics anymore. In order to introduce a binding, use e.g. the ”fresh” primitive instead (potential source of incompatibilities).
- New tactics is_ind, is_const, is_proj, is_constructor for use in Ltac.
- New goal selectors. Sets of goals can be selected by listing integers ranges. Example: ”1,4-7,24: tac” focuses ”tac” on goals 1,4,5,6,7,24.
- For uniformity with ”destruct”/”induction” and for a more natural behavior, ”injection” can now work in place by activating option ”Structural Injection”. In this case, hypotheses are also put in the context in the natural left-to-right order and the hypothesis on which injection applies is cleared.
- Tactic ”contradiction” (hence ”easy”) now also solve goals with hypotheses of the form ”~True” or ”t<>t” (possible source of incompatibilities because of more successes in automation, but generally a more intuitive strategy).
- Option ”Injection On Proofs” was renamed ”Keep Proof Equalities”. When enabled, injection and inversion do not drop equalities between objects in Prop. Still disabled by default.
- New tactics ”notypeclasses refine” and ”simple notypeclasses refine” that disallow typeclass resolution when typechecking their argument, for use in typeclass hints.
- Integration of LtacProf, a profiler for Ltac.
- Reduction tactics now accept more fine-grained flags: iota is now a shorthand for the new flags match, fix and cofix.
- The ssreflect subterm selection algorithm is now accessible to tactic writers through the ssrmatching plugin.
- When used as an argument of an ltac function, ”auto” without ”with” nor ”using” clause now correctly uses only the core hint database by default.

Hints

- Revised the syntax of [Hint Cut] to follow standard notation for regexps.
- Hint Mode now accepts ”!” which means that the mode matches only if the argument’s head is not an evar (it goes under applications, casts, and scrutinees of matches and projections).
- Hints can now take an optional user-given pattern, used only by [typeclasses eauto] with the [Filtered Unification] option on.

Typeclasses

- Many new options and new engine based on the proof monad. The [typeclasses eauto] tactic is now a multi-goal, multi-success tactic. See reference manual for more information. It is planned to replace auto and eauto in the following version. The 8.5 resolution engine is still available to help solve compatibility issues.

Program

- The ”Shrink Obligations” flag now applies to all obligations, not only those solved by the automatic tactic.
- ”Shrink Obligations” is on by default and deprecated. Minor source of incompatibility for code relying on the precise arguments of obligations.

Notations

- ”Bind Scope” can once again bind ”Funclass” and ”Sortclass”.

General infrastructure

- New configurable warning system which can be controlled with the vernacular command ”Set Warnings”, or, under coqc/coqtop, with the flag ”-w”. In particular, the default is now that warnings are printed by coqc.
- In asynchronous mode, Coq is now capable of recovering from errors and continue processing the document.

Tools

- coqc accepts a -o option to specify the output file name
- coqtop accepts –print-version to print Coq and OCaml versions in easy to parse format
- Setting [Printing Dependent Evars Line] can be unset to disable the computation associated with printing the ”dependent evars: ” line in -emacs mode
- Removed the -verbose-compat-notations flag and the corresponding Set Verbose Compat vernacular, since these warnings can now be silenced or turned into errors using ”-w”.

XML protocol

- message format has changed, see dev/doc/changes.txt for more details.

Many bug fixes, minor changes and documentation improvements are not mentioned here.

3.6.4 Details of changes in 8.6

Kernel

- Fixed critical bug #5248 in VM long multiplication on 32-bit architectures. Was there only since 8.6beta1, so no stable release impacted.

Other bug fixes in universes, type class shelving,...

3.6.5 Details of changes in 8.6.1

- Fix #5380: Default colors for CoqIDE are actually applied.
- Fix plugin warnings
- Document named evars (including Show ident)
- Fix Bug #5574, document function scope
- Adding a test case as requested in bug 5205.
- Fix Bug #5568, no dup notation warnings on repeated module imports
- Fix documentation of Typeclasses eauto :=
- Refactor documentation of records.

- Protecting from warnings while compiling 8.6
- Fixing an inconsistency between configure and configure.ml
- Add test-suite checks for coqchk with constraints
- Fix bug #5019 (looping zify on dependent types)
- Fix bug 5550: ”typeclasses eauto with” does not work with section variables.
- Bug 5546, qualify datatype constructors when needed in Show Match
- Bug #5535, test for Show with -emacs
- Fix bug #5486, don’t reverse ids in tuples
- Fixing #5522 (anomaly with free vars of pat)
- Fix bug #5526, don’t check for nonlinearity in notation if printing only
- Fix bug #5255
- Fix bug #3659: -time should understand multibyte encodings.
- FIx bug #5300: Anomaly: Uncaught exception Not_found” in ”Print Assumptions”.
- Fix outdated description in RefMan.
- Repairing `Set Rewriting Schemes`
- Fixing #5487 (v8.5 regression on ltac-matching expressions with evars).
- Fix description of command-line arguments for Add (Rec) LoadPath
- Fix bug #5377: @? patterns broken.
- add XML protocol doc
- Fix anomaly when doing [all:Check _.] during a proof.
- Correction of bug #4306
- Fix #5435: [Eval native_compute in] raises anomaly.
- Instances should obey universe binders even when defined by tactics.
- Intern names bound in match patterns
- funind: Ignore missing info for current function
- Do not typecheck twice the type of opaque constants.
- show unused intro pattern warning
- [future] Be eager when ”chaining” already resolved future values.
- Opaque side effects
- Fix #5132: coq_makefile generates incorrect install goal
- Run non-tactic comands without resilient_command
- Univs: fix bug #5365, generation of u+k <= v constraints
- make `emit` tail recursive
- Don’t require printing-only notation to be productive
- Fix the way setoid_rewrite handles bindings.
- Fix for bug 5244 - set printing width ignored when given enough space

- Fix bug 4969, autoapply was not tagging shelved subgoals correctly

3.7 Version 8.5

3.7.1 Summary of changes

Coq version 8.5 contains the result of five specific long-term projects:

- A new asynchronous evaluation and compilation mode by Enrico Tassi with help from Bruno Barras and Carst Tankink.
- Full integration of the new proof engine by Arnaud Spiwack helped by Pierre-Marie Pédrot,
- Addition of conversion and reduction based on native compilation by Maxime Dénès and Benjamin Grégoire.
- Full universe polymorphism for definitions and inductive types by Matthieu Sozeau.
- An implementation of primitive projections with η-conversion bringing significant performance improvements when using records by Matthieu Sozeau.

The full integration of the proof engine, by Arnaud Spiwack and Pierre-Marie Pédrot, brings to primitive tactics and the user level Ltac language dependent subgoals, deep backtracking and multiple goal handling, along with miscellaneous features and an improved potential for future modifications. Dependent subgoals allow statements in a goal to mention the proof of another. Proofs of unsolved subgoals appear as existential variables. Primitive backtracking makes it possible to write a tactic with several possible outcomes which are tried successively when subsequent tactics fail. Primitives are also available to control the backtracking behavior of tactics. Multiple goal handling paves the way for smarter automation tactics. It is currently used for simple goal manipulation such as goal reordering.

The way Coq processes a document in batch and interactive mode has been redesigned by Enrico Tassi with help from Bruno Barras. Opaque proofs, the text between Proof and Qed, can be processed asynchronously, decoupling the checking of definitions and statements from the checking of proofs. It improves the responsiveness of interactive development, since proofs can be processed in the background. Similarly, compilation of a file can be split into two phases: the first one checking only definitions and statements and the second one checking proofs. A file resulting from the first phase – with the .vio extension – can be already Required. All .vio files can be turned into complete .vo files in parallel. The same infrastructure also allows terminating tactics to be run in parallel on a set of goals via the `par:` goal selector.

CoqIDE was modified to cope with asynchronous checking of the document. Its source code was also made separate from that of Coq, so that CoqIDE no longer has a special status among user interfaces, paving the way for decoupling its release cycle from that of Coq in the future.

Carst Tankink developed a Coq back-end for user interfaces built on Makarius Wenzel's Prover IDE framework (PIDE), like PIDE/jEdit (with help from Makarius Wenzel) or PIDE/Coqoon (with help from Alexander Faithfull and Jesper Bengtson). The development of such features was funded by the Paral-ITP French ANR project.

The full universe polymorphism extension was designed by Matthieu Sozeau. It conservatively extends the universes system and core calculus with definitions and inductive declarations parameterized by universes and constraints. It is based on a modification of the kernel architecture to handle constraint checking only, leaving the generation of constraints to the refinement/type inference engine. Accordingly, tactics are now fully universe aware, resulting in more localized error messages in case of inconsistencies and allowing higher-level algorithms like unification to be entirely type safe. The internal representation of universes has been modified but this is invisible to the user.

The underlying logic has been extended with η-conversion for records defined with primitive projections by Matthieu Sozeau. This additional form of η-conversion is justified using the same principle than the previously added η-conversion for function types, based on formulations of the Calculus of Inductive Constructions with typed equality. Primitive projections, which do not carry the parameters of the record and are rigid names (not defined as a pattern matching construct), make working with nested records more manageable in terms of time and space consumption. This extension and universe polymorphism were carried out partly while Matthieu Sozeau was working at the IAS in Princeton.

The guard condition has been made compliant with extensional equality principles such as propositional extensionality and univalence, thanks to Maxime Dénès and Bruno Barras. To ensure compatibility with the univalence axiom, a new flag `-indices-matter` has been implemented, taking into account the universe levels of indices when computing the levels of inductive types. This supports using Coq as a tool to explore the relations between homotopy theory and type theory.

Maxime Dénès and Benjamin Grégoire developed an implementation of conversion test and normal form computation using the OCaml native compiler. It complements the virtual machine conversion offering much faster computation for expensive functions.

Coq 8.5 also comes with a bunch of many various smaller-scale changes and improvements regarding the different components of the system. We shall only list a few of them.

Pierre Boutillier developed an improved tactic for simplification of expressions called *cbn*.

Maxime Dénès maintained the bytecode-based reduction machine. Pierre Letouzey maintained the extraction mechanism.

Pierre-Marie Pédrot has extended the syntax of terms to, experimentally, allow holes in terms to be solved by a locally specified tactic.

Existential variables are referred to by identifiers rather than mere numbers, thanks to Hugo Herbelin who also improved the tactic language here and there.

Error messages for universe inconsistencies have been improved by Matthieu Sozeau. Error messages for unification and type inference failures have been improved by Hugo Herbelin, Pierre-Marie Pédrot and Arnaud Spiwack.

Pierre Courtieu contributed new features for using Coq through Proof General and for better interactive experience (bullets, Search, etc).

The efficiency of the whole system has been significantly improved thanks to contributions from Pierre-Marie Pédrot.

A distribution channel for Coq packages using the OPAM tool has been initiated by Thomas Braibant and developed by Guillaume Claret, with contributions by Enrico Tassi and feedback from Hugo Herbelin.

Packaging tools were provided by Pierre Letouzey and Enrico Tassi (Windows), Pierre Boutillier, Matthieu Sozeau and Maxime Dénès (MacOS X). Maxime Dénès improved significantly the testing and benchmarking support.

Many power users helped to improve the design of the new features via the bug tracker, the coq development mailing list or the Coq-Club mailing list. Special thanks are going to the users who contributed patches and intensive brain-storming, starting with Jason Gross, Jonathan Leivent, Greg Malecha, Clément Pit-Claudel, Marc Lasson, Lionel Rieg. It would however be impossible to mention with precision all names of people who to some extent influenced the development.

Version 8.5 is one of the most important releases of Coq. Its development spanned over about 3 years and a half with about one year of beta-testing. General maintenance during part or whole of this period has been done by Pierre Boutillier, Pierre Courtieu, Maxime Dénès, Hugo Herbelin, Pierre Letouzey, Guillaume Melquiond, Pierre-Marie Pédrot, Matthieu Sozeau, Arnaud Spiwack, Enrico Tassi as well as Bruno Barras, Yves Bertot, Frédéric Besson, Xavier Clerc, Pierre Corbineau, Jean-Christophe Filliâtre, Julien Forest, Sébastien Hinderer, Assia Mahboubi, Jean-Marc Notin, Yann Régis-Gianas, François Ripault, Carst Tankink. Maxime Dénès coordinated the release process.

Paris, January 2015, revised December 2015,
Hugo Herbelin, Matthieu Sozeau and the Coq development team

3.7.2 Potential sources of incompatibilities

List of typical changes to be done to adapt files from Coq 8.4 to Coq 8.5 when not using compatibility option `-compat 8.4`.

- Symptom: ”The reference omega was not found in the current environment”.

  Cause: ”Require Omega” does not import the tactic ”omega” any more

  Possible solutions:

  - use ”Require Import OmegaTactic” (not compatible with 8.4)
  - use ”Require Import Omega” (compatible with 8.4)
  - add definition ”Ltac omega := Coq.omega.Omega.omega.”

- Symptom: ”intuition” cannot solve a goal (not working anymore on non standard connective)

  Cause: ”intuition” had an accidental non uniform behavior fixed on non standard connectives

  Possible solutions:

  - use ”dintuition” instead; it is stronger than ”intuition” and works uniformly on non standard connectives, such as n-ary conjunctions or disjunctions (not compatible with 8.4)
  - do the script differently

- Symptom: The constructor foo (in type bar) expects n arguments.

  Cause: parameters must now be given in patterns

  Possible solutions:

  - use option ”Set Asymmetric Patterns” (compatible with 8.4)
  - add ”_” for the parameters (not compatible with 8.4)
  - turn the parameters into implicit arguments (compatible with 8.4)

- Symptom: ”NPeano.Nat.foo” not existing anymore

  Possible solutions:

  - use ”Nat.foo” instead

  Symptom: typing problems with proj1_sig or similar

  Cause: coercion from sig to sigT and similar coercions have been removed so as to make the initial state easier to understand for beginners

  Solution: change proj1_sig into projT1 and similarly (compatible with 8.4)

Other detailed changes

- options for *coq* compilation (see below for ocaml).

  - [-I foo] is now deprecated and will not add directory foo to the coq load path (only for ocaml, see below). Just replace [-I foo] by [-Q foo ""] in your project file and re-generate makefile. Or perform the same operation directly in your makefile if you edit it by hand.
  - Option -R Foo bar is the same in v8.5 than in v8.4 concerning coq load path.
  - Option [-I foo -as bar] is unchanged but discouraged unless you compile ocaml code. Use -Q foo bar instead.

  for more details: see section "Customization at launch time" of the reference manual.

- Command line options for ocaml Compilation of ocaml code (plugins)
  - [-I foo] is *not* deprecated to add foo to the ocaml load path.
  - [-I foo -as bar] adds foo to the ocaml load path *and* adds foo to the coq load path with logical name bar (shortcut for -I foo -Q foo bar).

  for more details: section "Customization at launch time" of the reference manual.

- Universe Polymorphism.
- Refinement, unification and tactics are now aware of universes, resulting in more localized errors. Universe inconsistencies should no more get raised at Qed time but during the proof. Unification *always* produces well-typed substitutions, hence some rare cases of unifications that succeeded while producing ill-typed terms before will now fail.
- The [change p with c] tactic semantics changed, now typechecking [c] at each matching occurrence [t] of the pattern [p], and converting [t] with [c].
- Template polymorphic inductive types: the partial application of a template polymorphic type (e.g. list) is not polymorphic. An explicit parameter application (e.g [fun A => list A]) or [apply (list _)] will result in a polymorphic instance.
- The type inference algorithm now takes opacity of constants into account. This may have effects on tactics using type inference (e.g. induction). Extra "Transparent" might have to be added to revert opacity of constants.

Type classes.

- When writing an `Instance foo : Class A := {| proj := t |}` (note the vertical bars), support for typechecking the projections using the type information and switching to proof mode is no longer available. Use `{ }` (without the vertical bars) instead.

Tactic abstract.

- Auxiliary lemmas generated by the abstract tactic are removed from the global environment and inlined in the proof term when a proof is ended with Qed. The behavior of 8.4 can be obtained by ending proofs with "Qed exporting" or "Qed exporting ident, .., ident".

3.7.3 Details of changes in 8.5beta1

Logic

- Primitive projections for records allow for a compact representation of projections, without parameters and avoid the behavior of defined projections that can unfold to a case expression. To turn the use of native projections on, use [Set Primitive Projections]. Record, Class and Structure types defined while this option is set will be defined with primitive projections instead of the usual encoding as a case expression. For compatibility, when p is a primitive projection, @p can be used to refer to the projection with explicit parameters, i.e. [@p] is definitionally equal to [λ params r. r.(p)]. Records with primitive projections have eta-conversion, the canonical form being [mkR pars (p1 t) ... (pn t)].

- New universe polymorphism (see reference manual)
- New option -type-in-type to collapse the universe hierarchy (this makes the logic inconsistent).
- The guard condition for fixpoints is now a bit stricter. Propagation of subterm value through pattern matching is restricted according to the return predicate. Restores compatibility of Coq's logic with the propositional extensionality axiom. May create incompatibilities in recursive programs heavily using dependent types.
- Trivial inductive types are no longer defined in Type but in Prop, which leads to a non-dependent induction principle being generated in place of the dependent one. To recover the old behavior, explicitly define your inductive types in Set.

Vernacular commands

- A command ”Variant” allows to define non-recursive variant types.
- The command ”Record foo ...” does not generate induction principles (foo_rect, foo_rec, foo_ind) anymore by default (feature wish #2693). The command ”Variant foo ...” does not either. A flag ”Set/Unset Nonrecursive Elimination Schemes” allows changing this. The tactic ”induction” on a ”Record” or a ”Variant” is now actually doing ”destruct”.
- The ”Open Scope” command can now be given also a delimiter (e.g. Z).
- The ”Definition” command now allows the ”Local” modifier, allowing for non-importable definitions. The same goes for ”Axiom” and ”Parameter”.
- Section-specific commands such as ”Let” (resp. ”Variable”, ”Hypothesis”) used out of a section now behave like the corresponding ”Local” command, i.e. ”Local Definition” (resp. ”Local Parameter”, ”Local Axiom”). (potential source of rare incompatibilities).
- The ”Let” command can now define local (co)fixpoints.
- Command ”Search” has been renamed into ”SearchHead”. The command name ”Search” now behaves like former ”SearchAbout”. The latter name is deprecated.
- ”Search”, ”About”, ”SearchHead”, ”SearchRewrite” and ”SearchPattern” now search for hypothesis (of the current goal by default) first. They now also support the goal selector prefix to specify another goal to search: e.g. ”n:Search id”. This is also true for SearchAbout although it is deprecated.
- The coq/user-contrib directory and the XDG directories are no longer recursively added to the load path, so files from installed libraries now need to be fully qualified for the ”Require” command to find them. The tools/update-require script can be used to convert a development.
- A new Print Strategies command allows visualizing the opacity status of the whole engine.
- The ”Locate” command now searches through all sorts of qualified namespaces of Coq: terms, modules, tactics, etc. The old behavior of the command can be retrieved using the ”Locate Term” command.
- New ”Derive” command to help writing program by derivation.
- New ”Refine Instance Mode” option that allows to deactivate the generation of obligations in incomplete typeclass instances, raising an error instead.
- ”Collection” command to name sets of section hypotheses. Named collections can be used in the syntax of ”Proof using” to assert which section variables are used in a proof.
- The ”Optimize Proof” command can be placed in the middle of a proof to force the compaction of the data structure used to represent the ongoing proof (evar map). This may result in a lower memory footprint and speed up the execution of the following tactics.
- ”Optimize Heap” command to tell the OCaml runtime to perform a major garbage collection step and heap compaction.

- `Instance` no longer treats the `{|...|}` syntax specially; it handles it in the same way as other commands, e.g. ”Definition”. Use the `{...}` syntax (no pipe symbols) to recover the old behavior.

Specification Language

- Slight changes in unification error messages.
- Added a syntax $(...)$ that allows putting tactics in terms (may break user notations using ”$(”, fixable by inserting a space or rewriting the notation).
- Constructors in pattern-matching patterns now respect the same rules regarding implicit arguments as in applicative position. The old behavior can be recovered by the command ”Set Asymmetric Patterns”. As a side effect, notations for constructors explicitly mentioning non-implicit parameters can now be used in patterns. Considering that the pattern language is already rich enough, binding local definitions is however now forbidden in patterns (source of incompatibilities for local definitions that delta-reduce to a constructor).
- Type inference algorithm now granting opacity of constants. This might also affect behavior of tactics (source of incompatibilities, solvable by re-declaring transparent constants which were set opaque).
- Existential variables are now referred to by an identifier and the relevant part of their instance is displayed by default. They can be reparsed. The naming policy is yet unstable and subject to changes in future releases.

Tactics

- New tactic engine allowing dependent subgoals, fully backtracking (also known as multiple success) tactics, as well as tactics which can consider multiple goals together. In the new tactic engine, instantiation information of existential variables is always propagated to tactics, removing the need to manually use the ”instantiate” tactics to mark propagation points.
  - New tactical (a+b) inserts a backtracking point. When (a+b);c fails during the execution of c, it can backtrack and try b instead of a.
  - New tactical (once a) removes all the backtracking points from a (i.e. it selects the first success of a).
  - Tactic ”constructor” is now fully backtracking. In case of incompatibilities (e.g. combinatoric explosion), the former behavior of ”constructor” can be retrieved by using instead ”[> once constructor ..]”. Thanks to backtracking, undocumented ”constructor <tac>” syntax is now equivalent to ”[> once (constructor; tac) ..]”.
  - New ”multimatch” variant of ”match” tactic which backtracks to new branches in case of a later failure. The ”match” tactic is equivalent to ”once multimatch”.
  - New selector ”all:” such that ”all:tac” applies tactic ”tac” to all the focused goals, instead of just the first one as is the default.
  - A corresponding new option Set Default Goal Selector ”all” makes the tactics in scripts be applied to all the focused goal by default
  - New selector ”par:” such that ”par:tac” applies the (terminating) tactic ”tac” to all the focused goal in parallel. The number of worker can be selected with -async-proofs-tac-j and also limited using the coqworkmgr utility.
  - New tactics ”revgoals”, ”cycle” and ”swap” to reorder goals.
  - The semantics of recursive tactics (introduced with ”Ltac t := ...” or ”let rec t := ... in ...”) changed slightly as t is now applied to every goal, not each goal independently. In particular it may be applied when no goals are left. This may cause tactics such as ”let rec t := constructor;t” to loop indefinitely. The simple fix is to rewrite the recursive calls as follows: ”let rec t := constructor;[t..]” which recovers the earlier behavior (source of rare incompatibilities).

  - New tactic language feature ”numgoals” to count number of goals. It is accompanied by a ”guard” tactic which fails if a Boolean test over integers does not pass.
  - New tactical ”[> ...]” to apply tactics to individual goals.
  - New tactic ”gfail” which works like ”fail” except it will also fail if every goal has been solved.
  - The refine tactic is changed not to use an ad hoc typing algorithm to generate subgoals. It also uses the dependent subgoal feature to generate goals to materialize every existential variable which is introduced by the refinement (source of incompatibilities).
  - A tactic shelve is introduced to manage the subgoals which may be solved by unification: shelve removes every goal it is applied to from focus. These goals can later be called back into focus by the Unshelve command.
  - A variant shelve_unifiable only removes those goals which appear as existential variables in other goals. To emulate the old refine, use ”refine c;shelve_unifiable”. This can still cause incompatibilities in rare occasions.
  - New ”give_up” tactic to skip over a goal. A proof containing given up goals cannot be closed with ”Qed”, but only with ”Admitted”.

- The implementation of the admit tactic has changed: no axiom is generated for the admitted sub proof. ”admit” is now an alias for ”give_up”. Code relying on this specific behavior of ”admit” can be made to work by:
  - Adding an ”Axiom” for each admitted subproof.
  - Adding a single ”Axiom proof_admitted : False.” and the Ltac definition ”Ltac admit := case proof_admitted.”.
- Matching using ”lazymatch” was fundamentally modified. It now behaves like ”match” (immediate execution of the matching branch) but without the backtracking mechanism in case of failure.
- New ”tryif t then u else v” tactical which executes ”u” in case of success of ”t” and ”v” in case of failure.
- New conversion tactic ”native_compute”: evaluates the goal (or an hypothesis) with a call-by-value strategy, using the OCaml native compiler. Useful on very intensive computations.
- New ”cbn” tactic, a well-behaved simpl.
- Repeated identical calls to omega should now produce identical proof terms.
- Tactics btauto, a reflexive Boolean tautology solver.
- Tactic ”tauto” was exceptionally able to destruct other connectives than the binary connectives ”and”, ”or”, ”prod”, ”sum”, ”iff”. This non-uniform behavior has been fixed (bug #2680) and tauto is slightly weaker (possible source of incompatibilities). On the opposite side, new tactic ”dtauto” is able to destruct any record-like inductive types, superseding the old version of ”tauto”.
- Similarly, ”intuition” has been made more uniform and, where it now fails, ”dintuition” can be used (possible source of incompatibilities).
- New option ”Unset Intuition Negation Unfolding” for deactivating automatic unfolding of ”not” in intuition.
- Tactic notations can now be defined locally to a module (use ”Local” prefix).
- Tactic ”red” now reduces head beta-iota redexes (potential source of rare incompatibilities).
- Tactic ”hnf” now reduces inner beta-iota redexes (potential source of rare incompatibilities).
- Tactic ”intro H” now reduces beta-iota redexes if these hide a product (potential source of rare incompatibilities).

- In Ltac matching on patterns of the form "_ pat1 ... patn" now behaves like if matching on "?X pat1 ... patn", i.e. accepting "_" to be instantiated by an applicative term (experimental at this stage, potential source of incompatibilities).
- In Ltac matching on goal, types of hypotheses are now interpreted in the %type scope (possible source of incompatibilities).
- "change ... in ..." and "simpl ... in ..." now properly consider nested occurrences (possible source of incompatibilities since this alters the numbering of occurrences), but do not support nested occurrences.
- Tactics simpl, vm_compute and native_compute can be given a notation string to a constant as argument.
- When given a reference as argument, simpl, vm_compute and native_compute now strictly interpret it as the head of a pattern starting with this reference.
- The "change p with c" tactic semantics changed, now type-checking "c" at each matching occurrence "t" of the pattern "p", and converting "t" with "c".
- Now "appcontext" and "context" behave the same. The old buggy behavior of "context" can be retrieved at parse time by setting the "Tactic Compat Context" flag (possible source of incompatibilities).
- New introduction pattern p/c which applies lemma c on the fly on the hypothesis under consideration before continuing with introduction pattern p.
- New introduction pattern [= x1 .. xn] applies "injection as [x1 .. xn]" on the fly if injection is applicable to the hypothesis under consideration (idea borrowed from Georges Gonthier). Introduction pattern [=] applies "discriminate" if a discriminable equality.
- New introduction patterns * and ** to respectively introduce all forthcoming dependent variables and all variables/hypotheses dependent or not.
- Tactic "injection c as ipats" now clears c if c refers to an hypothesis and moves the resulting equations in the hypotheses independently of the number of ipats, which has itself to be less than the number of new hypotheses (possible source of incompatibilities; former behavior obtainable by "Unset Injection L2R Pattern Order").
- Tactic "injection" now automatically simplifies subgoals "existT n p = existT n p'" into "p = p'" when "n" is in an inductive type for which a decidable equality scheme has been generated with "Scheme Equality" (possible source of incompatibilities).
- New tactic "rewrite_strat" for generalized rewriting with user-defined strategies, subsuming autorewrite.
- Injection can now also deduce equality of arguments of sort Prop, by using the option "Set Injection On Proofs" (disabled by default). Also improved the error messages.
- Tactic "subst id" now supports id occurring in dependent local definitions.
- Bugs fixed about intro-pattern "*" might lead to some rare incompatibilities.
- New tactical "time" to display time spent executing its argument.
- Tactics referring or using a constant dependent in a section variable which has been cleared or renamed in the current goal context now fail (possible source of incompatibilities solvable by avoiding clearing the relevant hypotheses).
- New construct "uconstr:c" and "type_term c" to build untyped terms.
- Binders in terms defined in Ltac (either "constr" or "uconstr") can now take their names from identifiers defined in Ltac. As a consequence, a name cannot be used in a binder "constr:(fun x => ...)" if an Ltac variable of that name already exists and does not contain an identifier. Source of occasional incompatibilities.

- The ”refine” tactic now accepts untyped terms built with ”uconstr” so that terms with holes can be constructed piecewise in Ltac.
- New bullets –, ++, , —, +++, *, ... made available.
- More informative messages when wrong bullet is used.
- Bullet suggestion when a subgoal is solved.
- New tactic ”enough”, symmetric to ”assert”, but with subgoals swapped, as a more friendly replacement of ”cut”.
- In destruct/induction, experimental modifier ”!” prefixing the hypothesis name to tell not erasing the hypothesis.
- Bug fixes in ”inversion as” may occasionally lead to incompatibilities.
- Behavior of introduction patterns -> and <- made more uniform (hypothesis is cleared, rewrite in hypotheses and conclusion and erasing the variable when rewriting a variable).
- New experimental option ”Set Standard Proposition Elimination Names” so that case analysis or induction on schemes in Type containing propositions now produces ”H”-based names.
- Tactics from plugins are now active only when the corresponding module is imported (source of incompatibilities, solvable by adding an ”Import”; in the particular case of Omega, use ”Require Import OmegaTactic”).
- Semantics of destruct/induction has been made more regular in some edge cases, possibly leading to incompatibilities:
  - new goals are now opened when the term does not match a subterm of the goal and has unresolved holes, while in 8.4 these holes were turned into existential variables
  - when no ”at” option is given, the historical semantics which selects all subterms syntactically identical to the first subterm matching the given pattern is used
  - non-dependent destruct/induction on an hypothesis with premises in an inductive type with indices is fixed
  - residual local definitions are now correctly removed.
- The rename tactic may now replace variables in parallel.
- A new ”Info” command replaces the ”info” tactical discontinued in v8.4. It still gives informative results in many cases.
- The ”info_auto” tactic is known to be broken and does not print a trace anymore. Use ”Info 1 auto” instead. The same goes for ”info_trivial”. On the other hand ”info_eauto” still works fine, while ”Info 1 eauto” prints a trivial trace.
- When using a lemma of the prototypical form ”forall A, {a:A & P a}”, ”apply” and ”apply in” do not instantiate anymore ”A” with the current goal and use ”a” as the proof, as they were sometimes doing, now considering that it is a too powerful decision.

Program

- ”Solve Obligations using” changed to ”Solve Obligations with”, consistent with ”Proof with”.
- Program Lemma, Definition now respect automatic introduction.
- Program Lemma, Definition, etc.. now interpret ”->” like Lemma and Definition as a non-dependent arrow (potential source of incompatibility).
- Add/document ”Set Hide Obligations” (to hide obligations in the final term inside an implicit argument) and ”Set Shrink Obligations” (to minimize dependencies of obligations defined by tactics).

Notations

- The syntax ”x -> y” is now declared at level 99. In particular, it has now a lower priority than ”<->”: ”A -> B <-> C” is now ”A -> (B <-> C)” (possible source of incompatibilities)
- Notations accept term-providing tactics using the $(...)$ syntax.
- ”Bind Scope” can no longer bind ”Funclass” and ”Sortclass”.
- A notation can be given a (compat ”8.x”) annotation, making it behave like a ”only parsing” notation, but the annotation may lead to eventually issue warnings or errors in further versions when this notation is used.
- More systematic insertion of spaces as a default for printing notations (”format” still available to override the default).
- In notations, a level modifier referring to a non-existent variable is now considered an error rather than silently ignored.

Tools

- Option -I now only adds directories to the ml path.
- Option -Q behaves as -R, except that the logical path of any loaded file has to be fully qualified.
- Option -R no longer adds recursively to the ml path; only the root directory is added. (Behavior with respect to the load path is unchanged.)
- Option -nois prevents coq/theories and coq/plugins to be recursively added to the load path. (Same behavior as with coq/user-contrib.)
- coqdep accepts a -dumpgraph option generating a dot file.
- Makefiles generated through coq_makefile have three new targets ”quick” ”checkproofs” and ”vio2vo”, allowing respectively to asynchronously compile the files without playing the proof scripts, asynchronously checking that the quickly generated proofs are correct and generating the object files from the quickly generated proofs.
- The XML plugin was discontinued and removed from the source.
- A new utility called coqworkmgr can be used to limit the number of concurrent workers started by independent processes, like make and CoqIDE. This is of interest for users of the par: goal selector.

Interfaces

- CoqIDE supports asynchronous edition of the document, ongoing tasks and errors are reported in the bottom right window. The number of workers taking care of processing proofs can be selected with -async-proofs-j.
- CoqIDE highlights in yellow ”unsafe” commands such as axiom declarations, and tactics like ”give_up”.
- CoqIDE supports Proof General like key bindings; to activate the PG mode go to Edit -> Preferences -> Editor. For the documentation see Help -> Help for PG mode.
- CoqIDE automatically retracts the locked area when one edits the locked text.
- CoqIDE search and replace got regular expressions power. See the documentation of OCaml’s Str module for the supported syntax.
- Many CoqIDE windows, including the query one, are now detachable to improve usability on multi screen work stations.
- Coqtop/coqc outputs highlighted syntax. Colors can be configured thanks to the COQ_COLORS environment variable, and their current state can be displayed with the -list-tags command line option.

- Third party user interfaces can install their main loop in $COQLIB/toploop and call coqtop with the -toploop flag to select it.

Internal Infrastructure

- Many reorganizations in the ocaml source files. For instance, many internal a.s.t. of Coq are now placed in mli files in a new directory intf/, for instance constrexpr.mli or glob_term.mli. More details in dev/doc/changes.
- The file states/initial.coq does not exist anymore. Instead, coqtop initially does a "Require" of Prelude.vo (or nothing when given the options -noinit or -nois).
- The format of vo files has slightly changed: cf final comments in checker/cic.mli.
- The build system does not produce anymore programs named coqtop.opt and a symbolic link to coqtop. Instead, coqtop is now directly an executable compiled with the best OCaml compiler available. The bytecode program coqtop.byte is still produced. Same for other utilities.
- Some options of the ./configure script slightly changed:
  - The -coqrunbyteflags and its blank-separated argument is replaced by option -vmbyteflags which expects a comma-separated argument.
  - The -coqtoolsbyteflags option is discontinued, see -no-custom instead.

Miscellaneous

- ML plugins now require a "DECLARE PLUGIN "foo"" statement. The "foo" name must be exactly the name of the ML module that will be loaded through a "Declare ML "foo"" command.

3.7.4 Details of changes in 8.5beta2

Logic

- The VM now supports inductive types with up to 8388851 non-constant constructors and up to 8388607 constant ones.

Specification language

- Syntax "$(tactic)$" changed to "ltac: tactic".

Tactics

- A script using the admit tactic can no longer be concluded by either Qed or Defined. In the first case, Admitted can be used instead. In the second case, a subproof should be used.
- The easy tactic and the now tactical now have a more predictable behavior, but they might now discharge some previously unsolved goals.

Extraction

- Definitions extracted to Haskell GHC should no longer randomly segfault when some Coq types cannot be represented by Haskell types.
- Definitions can now be extracted to Json for post-processing.

Tools

- Option -I -as has been removed, and option -R -as has been deprecated. In both cases, option -R can be used instead.
- coq_makefile now generates double-colon rules for rules such as clean.

API

- The interface of [change] has changed to take a [change_arg], which can be built from a [constr] using [make_change_arg].

3.7.5 Details of changes in 8.5beta3

Vernacular commands

- New command ”Redirect” to redirect the output of a command to a file.
- New command ”Undelimit Scope” to remove the delimiter of a scope.
- New option ”Strict Universe Declaration”, set by default. It enforces the declaration of all polymorphic universes appearing in a definition when introducing it.
- New command ”Show id” to show goal named id.
- Option ”Virtual Machine” removed.

Tactics

- New flag ”Regular Subst Tactic” which fixes ”subst” in situations where it failed to substitute all substitutable equations or failed to simplify cycles, or accidentally unfolded local definitions (flag is off by default).
- New flag ”Loose Hint Behavior” to handle hints loaded but not imported in a special way. It accepts three distinct flags: * ”Lax”, which is the default one, sets the old behavior, i.e. a non-imported hint behaves the same as an imported one. * ”Warn” outputs a warning when a non-imported hint is used. Note that this is an over-approximation, because a hint may be triggered by an eauto run that will eventually fail and backtrack. * ”Strict” changes the behavior of an unloaded hint to the one of the fail tactic, allowing to emulate the hopefully future import-scoped hint mechanism.
- New compatibility flag ”Universal Lemma Under Conjunction” which let tactics working under conjunctions apply sublemmas of the form ”forall A, ... -> A”.
- New compatibility flag ”Bracketing Last Introduction Pattern” which can be set so that the last disjunctive-conjunctive introduction pattern given to ”intros” automatically complete the introduction of its subcomponents, as the the disjunctive-conjunctive introduction patterns in non-terminal position already do.
- New flag ”Shrink Abstract” that minimalizes proofs generated by the abstract tactical w.r.t. variables appearing in the body of the proof.

Program

- The ”Shrink Obligations” flag now applies to all obligations, not only those solved by the automatic tactic.
- Importing Program no longer overrides the ”exists” tactic (potential source of incompatibilities).
- Hints costs are now correctly taken into account (potential source of incompatibilities).
- Documented the Hint Cut command that allows control of the proof-search during typeclass resolution (see reference manual).

API

- Some functions from pretyping/typing.ml and their derivatives were potential source of evarmap leaks, as they dropped their resulting evarmap. The situation was clarified by renaming them according to a `unsafe_*` scheme. Their sound variant is likewise renamed to their old name. The following renamings were made.
  - `Typing.type_of` -> `unsafe_type_of`

  - `Typing.e_type_of` -> `type_of`
  - A new `e_type_of` function that matches the `e_` prefix policy
  - `Tacmach.pf_type_of` -> `pf_unsafe_type_of`
  - A new safe `pf_type_of` function.

  All uses of `unsafe_*` functions should be eventually eliminated.

Tools

- Added an option -w to control the output of coqtop warnings.
- Configure now takes an optional -native-compiler (yes|no) flag replacing -no-native-compiler. The new flag is set to no by default under Windows.
- Flag -no-native-compiler was removed and became the default for coqc. If precompilation of files for native conversion test is desired, use -native-compiler.
- The -compile command-line option now takes the full path of the considered file, including the ".v" extension, and outputs a warning if such an extension is lacking.
- The -require and -load-vernac-object command-line options now take a logical path of a given library rather than a physical path, thus they behave like Require [Import] path.
- The -vm command-line option has been removed.

Standard Library

- There is now a Coq.Compat.Coq84 library, which sets the various compatibility options and does a few redefinitions to make Coq behave more like Coq v8.4. The standard way of putting Coq in v8.4 compatibility mode is to pass the command line flags "-require Coq.Compat.Coq84 -compat 8.4".

3.7.6 Details of changes in 8.5

Tools

- Flag "-compat 8.4" now loads Coq.Compat.Coq84. The standard way of putting Coq in v8.4 compatibility mode is to pass the command line flag "-compat 8.4". It can be followed by "-require Coq.Compat.AdmitAxiom" if the 8.4 behavior of admit is needed, in which case it uses an axiom.

Specification language

- Syntax "$(tactic)$" changed to "ltac:(tactic)".

Tactics

- Syntax "destruct !hyp" changed to "destruct (hyp)", and similarly for induction (rare source of incompatibilities easily solvable by removing parentheses around "hyp" when not for the purpose of keeping the hypothesis).
- Syntax "p/c" for on-the-fly application of a lemma c before introducing along pattern p changed to p%c1..%cn. The feature and syntax are in experimental stage.
- "Proof using" does not clear unused section variables.
- Tactic "refine" has been changed back to the 8.4 behavior of shelving subgoals that occur in other subgoals. The "refine" tactic of 8.5beta3 has been renamed "simple refine"; it does not shelve any subgoal.
- New tactical "unshelve tac" which grab existential variables put on the tactic shelve by the execution of "tac".

3.7.7 Details of changes in 8.5pl1

Critical bugfix

- The subterm relation for the guard condition was incorrectly defined on primitive projections (#4588)

Plugin development tools

- add a .merlin target to the makefile

Various performance improvements (time, space used by .vo files)

Other bugfixes

- Fix order of arguments to Big.compare_case in ExtrOcamlZBigInt.v
- Added compatibility coercions from Specif.v which were present in Coq 8.4.
- Fixing a source of inefficiency and an artificial dependency in the printer in the congruence tactic.
- Allow to unset the refinement mode of Instance in ML
- Fixing an incorrect use of prod_appvect on a term which was not a product in setoid_rewrite.
- Add -compat 8.4 econstructor tactics, and tests
- Add compatibility Nonrecursive Elimination Schemes
- Fixing the ”No applicable tactic” non informative error message regression on apply.
- Univs: fix get_current_context (bug #4603, part I)
- Fix a bug in Program coercion code
- Fix handling of arity of definitional classes.
- #4630: Some tactics are 20x slower in 8.5 than 8.4.
- #4627: records with no declared arity can be template polymorphic.
- #4623: set tactic too weak with universes (regression)
- Fix incorrect behavior of CS resolution
- #4591: Uncaught exception in directory browsing.
- CoqIDE is more resilient to initialization errors.
- #4614: ”Fully check the document” is uninterruptible.
- Try eta-expansion of records only on non-recursive ones
- Fix bug when a sort is ascribed to a Record
- Primitive projections: protect kernel from erroneous definitions.
- Fixed bug #4533 with previous Keyed Unification commit
- Win: kill unreliable hence do not waitpid after kill -9 (Close #4369)
- Fix strategy of Keyed Unification
- #4608: Anomaly ”output_value: abstract value (outside heap)”.
- #4607: do not read native code files if native compiler was disabled.
- #4105: poor escaping in the protocol between CoqIDE and coqtop.
- #4596: [rewrite] broke in the past few weeks.
- #4533 (partial): respect declared global transparency of projections in unification.ml

- #4544: Backtrack on using full betaiota reduction during keyed unification.
- #4540: CoqIDE bottom progress bar does not update.
- Fix regression from 8.4 in reflexivity
- #4580: [Set Refine Instance Mode] also used for Program Instance.
- #4582: cannot override notation [x]. MAY CREATE INCOMPATIBILITIES, see #4683.
- STM: Print/Extraction have to be skipped if -quick
- #4542: CoqIDE: STOP button also stops workers
- STM: classify some variants of Instance as regular `` `Fork` ` nodes.
- #4574: Anomaly: Uncaught exception Invalid_argument("splay_arity").
- Do not give a name to anonymous evars anymore. See bug #4547.
- STM: always stock in vio files the first node (state) of a proof
- STM: not delegate proofs that contain Vernac(Module|Require|Import), #4530
- Don't fail fatally if PATH is not set.
- #4537: Coq 8.5 is slower in typeclass resolution.
- #4522: Incorrect "Warning..." on windows.
- #4373: coqdep does not know about .vio files.
- #3826: "Incompatible module types" is uninformative.
- #4495: Failed assertion in metasyntax.ml.
- #4511: evar tactic can create non-typed evars.
- #4503: mixing universe polymorphic and monomorphic variables and definitions in sections is unsupported.
- #4519: oops, global shadowed local universe level bindings.
- #4506: Anomaly: File "pretyping/indrec.ml", line 169, characters 14-20: Assertion failed.
- #4548: Coqide crashes when going back one command

3.7.8 Details of changes in 8.5pl2

Critical bugfix

- Checksums of .vo files dependencies were not correctly checked.
- Unicode-to-ASCII translation was not injective, leading in a soundness bug in the native compiler.

Other bugfixes

- #4097: more efficient occur-check in presence of primitive projections
- #4398: type_scope used consistently in "match goal".
- #4450: eauto does not work with polymorphic lemmas
- #4677: fix alpha-conversion in notations needing eta-expansion.
- Fully preserve initial order of hypotheses in "Regular Subst Tactic" mode.
- #4644: a regression in unification.

- #4725: Function (Error: Conversion test raised an anomaly) and Program (Error: Cannot infer this placeholder of type)
- #4747: Problem building Coq 8.5pl1 with OCaml 4.03.0: Fatal warnings
- #4752: CoqIDE crash on files not ended by ".v".
- #4777: printing inefficiency with implicit arguments
- #4818: "Admitted" fails due to undefined universe anomaly after calling "destruct"
- #4823: remote counter: avoid thread race on sockets
- #4841: -verbose flag changed semantics in 8.5, is much harder to use
- #4851: [nsatz] cannot handle duplicated hypotheses
- #4858: Anomaly: Uncaught exception Failure("hd"). Please report. in variant of nsatz
- #4880: [nsatz_compute] generates invalid certificates if given redundant hypotheses
- #4881: synchronizing "Declare Implicit Tactic" with backtrack.
- #4882: anomaly with Declare Implicit Tactic on hole of type with evars
- Fix use of "Declare Implicit Tactic" in refine. triggered by CoqIDE
- #4069, #4718: congruence fails when universes are involved.

Universes

- Disallow silently dropping universe instances applied to variables (forward compatible)
- Allow explicit universe instances on notations, when they can apply to the head reference of their expansion.

Build infrastructure

- New update on how to find camlp5 binary and library at configure time.

3.7.9 Details of changes in 8.5pl3

Critical bugfix

- #4876: Guard checker incompleteness when using primitive projections

Other bugfixes

- #4780: Induction with universe polymorphism on was creating ill-typed terms.
- #4673: regression in setoid_rewrite, unfolding let-ins for type unification.
- #4754: Regression in setoid_rewrite, allow postponed unification problems to remain.
- #4769: Anomaly with universe polymorphic schemes defined inside sections.
- #3886: Program: duplicate obligations of mutual fixpoints.
- #4994: Documentation typo.
- #5008: Use the "md5" command on OpenBSD.
- #5007: Do not assume the "TERM" environment variable is always set.
- #4606: Output a break before a list only if there was an empty line.
- #5001: metas not cleaned properly in clenv_refine_in.
- #2336: incorrect glob data for module symbols (bug #2336).

- #4832: Remove extraneous dot in error message.
- Anomaly in printing a unification error message.
- #4947: Options which take string arguments are not backwards compatible.
- #4156: micromega cache files are now hidden files.
- #4871: interrupting par:abstract kills coqtop.
- #5043: [Admitted] lemmas pick up section variables.
- Fix name of internal refine ("simple refine").
- #5062: probably a typo in Strict Proofs mode.
- #5065: Anomaly: Not a proof by induction.
- Restore native compiler optimizations, they were disabled since 8.5!
- #5077: failure on typing a fixpoint with evars in its type.
- Fix recursive notation bug.
- #5095: non relevant too strict test in let-in abstraction.
- Ensuring that the evar name is preserved by "rename".
- #4887: confusion between using and with in documentation of firstorder.
- Bug in subst with let-ins.
- #4762: eauto weaker than auto.
- Remove if_then_else (was buggy). Use tryif instead.
- #4970: confusion between special "{" and non special "{{" in notations.
- #4529: primitive projections unfolding.
- #4416: Incorrect "Error: Incorrect number of goals".
- #4863: abstract in typeclass hint fails.
- #5123: unshelve can impact typeclass resolution
- Fix a collision about the meta-variable ".." in recursive notations.
- Fix printing of info_auto.
- #3209: Not_found due to an occur-check cycle.
- #5097: status of evars refined by "clear" in ltac: closed wrt evars.
- #5150: Missing dependency of the test-suite subsystems in prerequisite.
- Fix a bug in error printing of unif constraints
- #3941: Do not stop propagation of signals when Coq is busy.
- #4822: Incorrect assertion in cbn.
- #3479 parsing of "{" and "}" when a keyword starts with "{" or "}".
- #5127: Memory corruption with the VM.
- #5102: bullets parsing broken by calls to parse_entry.

Various documentation improvements

3.8 Version 8.4

3.8.1 Summary of changes

Coq version 8.4 contains the result of three long-term projects: a new modular library of arithmetic by Pierre Letouzey, a new proof engine by Arnaud Spiwack and a new communication protocol for CoqIDE by Vincent Gross.

The new modular library of arithmetic extends, generalizes and unifies the existing libraries on Peano arithmetic (types nat, N and BigN), positive arithmetic (type positive), integer arithmetic (Z and BigZ) and machine word arithmetic (type Int31). It provides with unified notations (e.g. systematic use of add and mul for denoting the addition and multiplication operators), systematic and generic development of operators and properties of these operators for all the types mentioned above, including gcd, pcm, power, square root, base 2 logarithm, division, modulo, bitwise operations, logical shifts, comparisons, iterators, ...

The most visible feature of the new proof engine is the support for structured scripts (bullets and proof brackets) but, even if yet not user-available, the new engine also provides the basis for refining existential variables using tactics, for applying tactics to several goals simultaneously, for reordering goals, all features which are planned for the next release. The new proof engine forced Pierre Letouzey to reimplement info and Show Script differently.

Before version 8.4, CoqIDE was linked to Coq with the graphical interface living in a separate thread. From version 8.4, CoqIDE is a separate process communicating with Coq through a textual channel. This allows for a more robust interfacing, the ability to interrupt Coq without interrupting the interface, and the ability to manage several sessions in parallel. Relying on the infrastructure work made by Vincent Gross, Pierre Letouzey, Pierre Boutillier and Pierre-Marie Pédrot contributed many various refinements of CoqIDE.

Coq 8.4 also comes with a bunch of various smaller-scale changes and improvements regarding the different components of the system.

The underlying logic has been extended with η-conversion thanks to Hugo Herbelin, Stéphane Glondu and Benjamin Grégoire. The addition of η-conversion is justified by the confidence that the formulation of the Calculus of Inductive Constructions based on typed equality (such as the one considered in Lee and Werner to build a set-theoretic model of CIC *[LW11]*) is applicable to the concrete implementation of Coq.

The underlying logic benefited also from a refinement of the guard condition for fixpoints by Pierre Boutillier, the point being that it is safe to propagate the information about structurally smaller arguments through β-redexes that are blocked by the “match” construction (blocked commutative cuts).

Relying on the added permissiveness of the guard condition, Hugo Herbelin could extend the pattern matching compilation algorithm so that matching over a sequence of terms involving dependencies of a term or of the indices of the type of a term in the type of other terms is systematically supported.

Regarding the high-level specification language, Pierre Boutillier introduced the ability to give implicit arguments to anonymous functions, Hugo Herbelin introduced the ability to define notations with several binders (e.g. `exists x y z, P`), Matthieu Sozeau made the typeclass inference mechanism more robust and predictable, Enrico Tassi introduced a command Arguments that generalizes Implicit Arguments and Arguments Scope for assigning various properties to arguments of constants. Various improvements in the type inference algorithm were provided by Matthieu Sozeau and Hugo Herbelin with contributions from Enrico Tassi.

Regarding tactics, Hugo Herbelin introduced support for referring to expressions occurring in the goal by pattern in tactics such as set or destruct. Hugo Herbelin also relied on ideas from Chung-Kil Hur’s Heq plugin to introduce automatic computation of occurrences to generalize when using destruct and induction on types with indices. Stéphane Glondu introduced new tactics *`constr_eq`*, *`is_evar`*, and *`has_evar`*, to be used when writing complex tactics. Enrico Tassi added support to fine-tuning the behavior of *`simpl`*. Enrico Tassi added the ability to specify over which variables of a section a lemma has to be exactly generalized.

Pierre Letouzey added a tactic timeout and the interruptibility of *vm_compute*. Bug fixes and miscellaneous improvements of the tactic language came from Hugo Herbelin, Pierre Letouzey and Matthieu Sozeau.

Regarding decision tactics, Loïc Pottier maintained nsatz, moving in particular to a typeclass based reification of goals while Frédéric Besson maintained Micromega, adding in particular support for division.

Regarding vernacular commands, Stéphane Glondu provided new commands to analyze the structure of type universes.

Regarding libraries, a new library about lists of a given length (called vectors) has been provided by Pierre Boutillier. A new instance of finite sets based on Red-Black trees and provided by Andrew Appel has been adapted for the standard library by Pierre Letouzey. In the library of real analysis, Yves Bertot changed the definition of π and provided a proof of the long-standing fact yet remaining unproved in this library, namely that $sin\frac{\pi}{2} = 1$.

Pierre Corbineau maintained the Mathematical Proof Language (C-zar).

Bruno Barras and Benjamin Grégoire maintained the call-by-value reduction machines.

The extraction mechanism benefited from several improvements provided by Pierre Letouzey.

Pierre Letouzey maintained the module system, with contributions from Élie Soubiran.

Julien Forest maintained the Function command.

Matthieu Sozeau maintained the setoid rewriting mechanism.

Coq related tools have been upgraded too. In particular, coq_makefile has been largely revised by Pierre Boutillier. Also, patches from Adam Chlipala for coqdoc have been integrated by Pierre Boutillier.

Bruno Barras and Pierre Letouzey maintained the `coqchk` checker.

Pierre Courtieu and Arnaud Spiwack contributed new features for using Coq through Proof General.

The Dp plugin has been removed. Use the plugin provided with Why 3 instead (http://why3.lri.fr/).

Under the hood, the Coq architecture benefited from improvements in terms of efficiency and robustness, especially regarding universes management and existential variables management, thanks to Pierre Letouzey and Yann Régis-Gianas with contributions from Stéphane Glondu and Matthias Puech. The build system is maintained by Pierre Letouzey with contributions from Stéphane Glondu and Pierre Boutillier.

A new backtracking mechanism simplifying the task of external interfaces has been designed by Pierre Letouzey.

The general maintenance was done by Pierre Letouzey, Hugo Herbelin, Pierre Boutillier, Matthieu Sozeau and Stéphane Glondu with also specific contributions from Guillaume Melquiond, Julien Narboux and Pierre-Marie Pédrot.

Packaging tools were provided by Pierre Letouzey (Windows), Pierre Boutillier (MacOS), Stéphane Glondu (Debian). Releasing, testing and benchmarking support was provided by Jean-Marc Notin.

Many suggestions for improvements were motivated by feedback from users, on either the bug tracker or the Coq-Club mailing list. Special thanks are going to the users who contributed patches, starting with Tom Prince. Other patch contributors include Cédric Auger, David Baelde, Dan Grayson, Paolo Herms, Robbert Krebbers, Marc Lasson, Hendrik Tews and Eelis van der Weegen.

Paris, December 2011
Hugo Herbelin

3.8.2 Potential sources of incompatibilities

The main known incompatibilities between 8.3 and 8.4 are consequences of the following changes:

- The reorganization of the library of numbers:

  Several definitions have new names or are defined in modules of different names, but a special care has been taken to have this renaming transparent for the user thanks to compatibility notations.

  However some definitions have changed, what might require some adaptations. The most noticeable examples are:

  - The ”?=” notation which now bind to Pos.compare rather than former Pcompare (now Pos.compare_cont).
  - Changes in names may induce different automatically generated names in proof scripts (e.g. when issuing ”destruct Z_le_gt_dec”).
  - Z.add has a new definition, hence, applying ”simpl” on subterms of its body might give different results than before.
  - BigN.shiftl and BigN.shiftr have reversed arguments order, the power function in BigN now takes two BigN.
- Other changes in libraries:
  - The definition of functions over ”vectors” (list of fixed length) have changed.
  - TheoryList.v has been removed.
- Slight changes in tactics:
  - Less unfolding of fixpoints when applying destruct or inversion on a fixpoint hiding an inductive type (add an extra call to simpl to preserve compatibility).
  - Less unexpected local definitions when applying ”destruct” (incompatibilities solvable by adapting name hypotheses).
  - Tactic ”apply” might succeed more often, e.g. by now solving pattern-matching of the form ?f x y = g(x,y) (compatibility ensured by using ”Unset Tactic Pattern Unification”), but also because it supports (full) betaiota (using ”simple apply” might then help).
  - Tactic autorewrite does no longer instantiate pre-existing existential variables.
  - Tactic ”info” is now available only for auto, eauto and trivial.
- Miscellaneous changes:
  - The command ”Load” is now atomic for backtracking (use ”Unset Atomic Load” for compatibility).

3.8.3 Details of changes in 8.4beta

Logic

- Standard eta-conversion now supported (dependent product only).
- Guard condition improvement: subterm property is propagated through beta-redex blocked by pattern-matching, as in ”(match v with C .. => fun x => u end) x”; this allows for instance to use ”rewrite ... in ...” without breaking the guard condition.

Specification language and notations

- Maximal implicit arguments can now be set locally by { }. The registration traverses fixpoints and lambdas. Because there is conversion in types, maximal implicit arguments are not taken into account in partial applications (use eta expanded form with explicit { } instead).
- Added support for recursive notations with binders (allows for instance to write ”exists x y z, P”).
- Structure/Record printing can be disable by ”Unset Printing Records”. In addition, it can be controlled on type by type basis using ”Add Printing Record” or ”Add Printing Constructor”.
- Pattern-matching compilation algorithm: in ”match x, y with ... end”, possible dependencies of x (or of the indices of its type) in the type of y are now taken into account.

Tactics

- New proof engine.
- Scripts can now be structured thanks to bullets - * + and to subgoal delimitation via { }. Note: for use with Proof General, a cvs version of Proof General no older than mid-July 2011 is currently required.
- Support for tactical ”info” is suspended.
- Support for command ”Show Script” is suspended.
- New tactics constr_eq, is_evar and has_evar for use in Ltac (DOC TODO).
- Removed the two-argument variant of ”decide equality”.
- New experimental tactical ”timeout <n> <tac>”. Since <n> is a time in second for the moment, this feature should rather be avoided in scripts meant to be machine-independent.
- Fix in ”destruct”: removal of unexpected local definitions in context might result in some rare incompatibilities (solvable by adapting name hypotheses).
- Introduction pattern ”_” made more robust.
- Tactic (and Eval command) vm_compute can now be interrupted via Ctrl-C.
- Unification in ”apply” supports unification of patterns of the form ?f x y = g(x,y) (compatibility ensured by using ”Unset Tactic Pattern Unification”). It also supports (full) betaiota.
- Tactic autorewrite does no longer instantiate pre-existing existential variables (theoretical source of possible incompatibilities).
- Tactic ”dependent rewrite” now supports equality in ”sig”.
- Tactic omega now understands Zpred (wish #1912) and can prove any goal from a context containing an arithmetical contradiction (wish #2236).
- Using ”auto with nocore” disables the use of the ”core” database (wish #2188). This pseudo-database ”nocore” can also be used with trivial and eauto.
- Tactics ”set”, ”destruct” and ”induction” accepts incomplete terms and use the goal to complete the pattern assuming it is non ambiguous.
- When used on arguments with a dependent type, tactics such as ”destruct”, ”induction”, ”case”, ”elim”, etc. now try to abstract automatically the dependencies over the arguments of the types (based on initial ideas from Chung-Kil Hur, extension to nested dependencies suggested by Dan Grayson)
- Tactic ”injection” now failing on an equality showing no constructors while it was formerly generalizing again the goal over the given equality.
- In Ltac, the ”context [...]” syntax has now a variant ”appcontext [...]” allowing to match partial applications in larger applications.

- When applying destruct or inversion on a fixpoint hiding an inductive type, recursive calls to the fixpoint now remain folded by default (rare source of incompatibility generally solvable by adding a call to simpl).
- In an ltac pattern containing a "match", a final "| _ => _" branch could be used now instead of enumerating all remaining constructors. Moreover, the pattern "match _ with _ => _ end" now allows to match any "match". A "in" annotation can also be added to restrict to a precise inductive type.
- The behavior of "simpl" can be tuned using the "Arguments" vernacular. In particular constants can be marked so that they are always/never unfolded by "simpl", or unfolded only when a set of arguments evaluates to a constructor. Last one can mark a constant so that it is unfolded only if the simplified term does not expose a match in head position.

Vernacular commands

- It is now mandatory to have a space (or tabulation or newline or end-of-file) after a "." ending a sentence.
- In SearchAbout, the [ ] delimiters are now optional.
- New command "Add/Remove Search Blacklist <substring> ...": a Search or SearchAbout or similar query will never mention lemmas whose qualified names contain any of the declared substrings. The default blacklisted substrings are `_subproof`, `Private_`.
- When the output file of "Print Universes" ends in ".dot" or ".gv", the universe graph is printed in the DOT language, and can be processed by Graphviz tools.
- New command "Print Sorted Universes".
- The undocumented and obsolete option "Set/Unset Boxed Definitions" has been removed, as well as syntaxes like "Boxed Fixpoint foo".
- A new option "Set Default Timeout n / Unset Default Timeout".
- Qed now uses information from the reduction tactics used in proof script to avoid conversion at Qed time to go into a very long computation.
- New command "Show Goal ident" to display the statement of a goal, even a closed one (available from Proof General).
- Command "Proof" accept a new modifier "using" to force generalization over a given list of section variables at section ending (DOC TODO).
- New command "Arguments" generalizing "Implicit Arguments" and "Arguments Scope" and that also allows to rename the parameters of a definition and to tune the behavior of the tactic "simpl".

Module System

- During subtyping checks, an opaque constant in a module type could now be implemented by anything of the right type, even if bodies differ. Said otherwise, with respect to subtyping, an opaque constant behaves just as a parameter. Coqchk was already implementing this, but not coqtop.
- The inlining done during application of functors can now be controlled more precisely, by the annotations (no inline) or (inline at level XX). With the latter annotation, only functor parameters whose levels are lower or equal than XX will be inlined. The level of a parameter can be fixed by "Parameter Inline(30) foo". When levels aren't given, the default value is 100. One can also use the flag "Set Inline Level ..." to set a level (DOC TODO).
- Print Assumptions should now handle correctly opaque modules (#2168).
- Print Module (Type) now tries to print more details, such as types and bodies of the module elements. Note that Print Module Type could be used on a module to display only its interface. The option "Set

Short Module Printing" could be used to switch back to the earlier behavior were only field names were displayed.

Libraries

- Extension of the abstract part of Numbers, which now provide axiomatizations and results about many more integer functions, such as pow, gcd, lcm, sqrt, log2 and bitwise functions. These functions are implemented for nat, N, BigN, Z, BigZ. See in particular file NPeano for new functions about nat.
- The definition of types positive, N, Z is now in file BinNums.v
- Major reorganization of ZArith. The initial file ZArith/BinInt.v now contains an internal module Z implementing the Numbers interface for integers. This module Z regroups:
  - all functions over type Z : Z.add, Z.mul, ...
  - the minimal proofs of specifications for these functions : Z.add_0_l, ...
  - an instantiation of all derived properties proved generically in Numbers : Z.add_comm, Z.add_assoc, ...

  A large part of ZArith is now simply compatibility notations, for instance Zplus_comm is an alias for Z.add_comm. The direct use of module Z is now recommended instead of relying on these compatibility notations.
- Similar major reorganization of NArith, via a module N in NArith/BinNat.v
- Concerning the positive datatype, BinPos.v is now in a specific directory PArith, and contains an internal submodule Pos. We regroup there functions such as Pos.add Pos.mul etc as well as many results about them. These results are here proved directly (no Number interface for strictly positive numbers).
- Note that in spite of the compatibility layers, all these reorganizations may induce some marginal incompatibilies in scripts. In particular:
  - the "?=" notation for positive now refers to a binary function Pos.compare, instead of the infamous ternary Pcompare (now Pos.compare_cont).
  - some hypothesis names generated by the system may changed (typically for a "destruct Z_le_gt_dec") since naming is done after the short name of the head predicate (here now "le" in module Z instead of "Zle", etc).
  - the internals of Z.add has changed, now relying of Z.pos_sub.
- Also note these new notations:
  - "<?" "<=?" "=?" for boolean tests such as Z.ltb Z.leb Z.eqb.
  - "÷" for the alternative integer division Z.quot implementing the Truncate convention (former ZOdiv), while the notation for the Coq usual division Z.div implementing the Flooring convention remains "/". Their corresponding modulo functions are Z.rem (no notations) for Z.quot and Z.modulo (infix "mod" notation) for Z.div.
- Lemmas about conversions between these datatypes are also organized in modules, see for instance modules Z2Nat, N2Z, etc.
- When creating BigN, the macro-generated part NMake_gen is much smaller. The generic part NMake has been reworked and improved. Some changes may introduce incompatibilities. In particular, the order of the arguments for BigN.shiftl and BigN.shiftr is now reversed: the number to shift now comes first. By default, the power function now takes two BigN.
- Creation of Vector, an independent library for lists indexed by their length. Vectors' names overwrite lists' one so you should not "Import" the library. All old names changed: function names follow

the ocaml ones and, for example, Vcons becomes Vector.cons. You can get [..;..;..]-style notations by importing Vector.VectorNotations.

- Removal of TheoryList. Requiring List instead should work most of the time.
- New syntax ”rew Heq in H” and ”rew <- Heq in H” for eq_rect and eq_rect_r (available by importing module EqNotations).
- Wf.iter_nat is now Peano.nat_iter (with an implicit type argument).

Internal infrastructure

- Opaque proofs are now loaded lazily by default. This allows to be almost as fast as -dont-load-proofs, while being safer (no creation of axioms) and avoiding feature restrictions (Print and Print Assumptions work ok).
- Revised hash-consing code allowing more sharing of memory
- Experimental support added for camlp4 (the one provided alongside ocaml), simply pass option -usecamlp4 to ./configure. By default camlp5 is used.
- Revised build system: no more stages in Makefile thanks to some recursive aspect of recent gnu make, use of vo.itarget files containing .v to compile for both make and ocamlbuild, etc.
- Support of cross-compilation via mingw from unix toward Windows, contact P. Letouzey for more informations.
- New Makefile rules mli-doc to make html of mli in dev/doc/html and full-stdlib to get a (huge) pdf reflecting the whole standard library.

Extraction

- By default, opaque terms are now truly considered opaque by extraction: instead of accessing their body, they are now considered as axioms. The previous behaviour can be reactivated via the option ”Set Extraction AccessOpaque”.
- The pretty-printer for Haskell now produces layout-independent code
- A new command ”Separate Extraction cst1 cst2 ...” that mixes a minimal extracted environment a la ”Recursive Extraction” and the production of several files (one per coq source) a la ”Extraction Library” (DOC TODO).
- New option ”Set/Unset Extraction KeepSingleton” for preventing the extraction to optimize singleton container types (DOC TODO).
- The extraction now identifies and properly rejects a particular case of universe polymorphism it cannot handle yet (the pair (I,I) being Prop).
- Support of anonymous fields in record (#2555).

CoqIDE

- Coqide now runs coqtop as separated process, making it more robust: coqtop subprocess can be interrupted, or even killed and relaunched (cf button ”Restart Coq”, ex-”Go to Start”). For allowing such interrupts, the Windows version of coqide now requires Windows >= XP SP1.
- The communication between CoqIDE and Coqtop is now done via a dialect of XML (DOC TODO).
- The backtrack engine of CoqIDE has been reworked, it now uses the ”Backtrack” command similarly to Proof General.
- The Coqide parsing of sentences has be reworked and now supports tactic delimitation via { }.
- Coqide now accepts the Abort command (wish #2357).

- Coqide can read coq_makefile files as ”project file” and use it to set automatically options to send to coqtop.
- Preference files have moved to $XDG_CONFIG_HOME/coq and accelerators are not stored as a list anymore.

Tools

- Coq now searches directories specified in COQPATH, $XDG_DATA_HOME/coq, $XDG_DATA_DIRS/coq, and user-contribs before the standard library.
- Coq rc file has moved to $XDG_CONFIG_HOME/coq.
- Major changes to coq_makefile:
  - mli/mlpack/mllib taken into account, ml not preproccessed anymore, ml4 work;
  - mlihtml generates doc of mli, install-doc install the html doc in DOCDIR with the same policy as vo in COQLIB;
  - More variables are given by coqtop -config, others are defined only if the users doesn’t have defined them elsewhere. Consequently, generated makefile should work directly on any architecture;
  - Packagers can take advantage of $(DSTROOT) introduction. Installation can be made in $XDG_DATA_HOME/coq;
  - -arg option allows to send option as argument to coqc.

3.8.4 Details of changes in 8.4beta2

Vernacular commands

- Commands ”Back” and ”BackTo” are now handling the proof states. They may perform some extra steps of backtrack to avoid states where the proof state is unavailable (typically a closed proof).
- The commands ”Suspend” and ”Resume” have been removed.
- A basic Show Script has been reintroduced (no indentation).
- New command ”Set Parsing Explicit” for deactivating parsing (and printing) of implicit arguments (useful for teaching).
- New command ”Grab Existential Variables” to transform the unresolved evars at the end of a proof into goals.

Tactics

- Still no general ”info” tactical, but new specific tactics info_auto, info_eauto, info_trivial which provides information on the proofs found by auto/eauto/trivial. Display of these details could also be activated by ”Set Info Auto”/”Set Info Eauto”/”Set Info Trivial”.
- Details on everything tried by auto/eauto/trivial during a proof search could be obtained by ”debug auto”, ”debug eauto”, ”debug trivial” or by a global ”Set Debug Auto”/”Set Debug Eauto”/”Set Debug Trivial”.
- New command ”r string” in Ltac debugger that interprets ”idtac string” in Ltac code as a breakpoint and jumps to its next use.
- Tactics from the Dp plugin (simplify, ergo, yices, cvc3, z3, cvcl, harvey, zenon, gwhy) have been removed, since Why2 has not been maintained for the last few years. The Why3 plugin should be a suitable replacement in most cases.

Libraries

- MSetRBT: a new implementation of MSets via Red-Black trees (initial contribution by Andrew Appel).
- MSetAVL: for maximal sharing with the new MSetRBT, the argument order of Node has changed (this should be transparent to regular MSets users).

Module System

- The names of modules (and module types) are now in a fully separated namespace from ordinary definitions: "Definition E:=0. Module E. End E." is now accepted.

CoqIDE

- Coqide now supports the "Restart" command, and "Undo" (with a warning). Better support for "Abort".

3.8.5 Details of changes in 8.4

Vernacular commands

- The "Reset" command is now supported again in files given to coqc or Load.
- "Show Script" now indents again the displayed scripts. It can also work correctly across Load'ed files if the option "Unset Atomic Load" is used.
- "Open Scope" can now be given the delimiter (e.g. Z) instead of the full scope name (e.g. Z_scope).

Notations

- Most compatibility notations of the standard library are now tagged as (compat xyz), where xyz is a former Coq version, for instance "8.3". These notations behave as (only parsing) notations, except that they may triggers warnings (or errors) when used while Coq is not in a corresponding -compat mode.
- To activate these compatibility warnings, use "Set Verbose Compat Notations" or the command-line flag -verbose-compat-notations.
- For a strict mode without these compatibility notations, use "Unset Compat Notations" or the command-line flag -no-compat-notations.

Tactics

- An annotation "eqn:H" or "eqn:?" can be added to a "destruct" or "induction" to make it generate equations in the spirit of "case_eq". The former syntax "_eqn" is discontinued.
- The name of the hypothesis introduced by tactic "remember" can be set via the new syntax "remember t as x eqn:H" (wish #2489).

Libraries

- Reals: changed definition of PI, no more axiom about sin(PI/2).
- SetoidPermutation: a notion of permutation for lists modulo a setoid equality.
- BigN: fixed the ocaml code doing the parsing/printing of big numbers.
- List: a couple of lemmas added especially about no-duplication, partitions.
- Init: Removal of the coercions between variants of sigma-types and subset types (possible source of incompatibility).

3.9 Version 8.3

3.9.1 Summary of changes

Coq version 8.3 is before all a transition version with refinements or extensions of the existing features and libraries and a new tactic nsatz based on Hilbert's Nullstellensatz for deciding systems of equations over rings.

With respect to libraries, the main evolutions are due to Pierre Letouzey with a rewriting of the library of finite sets FSets and a new round of evolutions in the modular development of arithmetic (library Numbers). The reason for making FSets evolve is that the computational and logical contents were quite intertwined in the original implementation, leading in some cases to longer computations than expected and this problem is solved in the new MSets implementation. As for the modular arithmetic library, it was only dealing with the basic arithmetic operators in the former version and its current extension adds the standard theory of the division, min and max functions, all made available for free to any implementation of $\mathbb{N}$, $\mathbb{Z}$ or $\mathbb{Z}/n\mathbb{Z}$.

The main other evolutions of the library are due to Hugo Herbelin who made a revision of the sorting library (including a certified merge-sort) and to Guillaume Melquiond who slightly revised and cleaned up the library of reals.

The module system evolved significantly. Besides the resolution of some efficiency issues and a more flexible construction of module types, Élie Soubiran brought a new model of name equivalence, the Δ-equivalence, which respects as much as possible the names given by the users. He also designed with Pierre Letouzey a new, convenient operator `<+` for nesting functor application that provides a light notation for inheriting the properties of cascading modules.

The new tactic nsatz is due to Loïc Pottier. It works by computing Gröbner bases. Regarding the existing tactics, various improvements have been done by Matthieu Sozeau, Hugo Herbelin and Pierre Letouzey.

Matthieu Sozeau extended and refined the typeclasses and Program features (the Russell language). Pierre Letouzey maintained and improved the extraction mechanism. Bruno Barras and Élie Soubiran maintained the Coq checker, Julien Forest maintained the Function mechanism for reasoning over recursively defined functions. Matthieu Sozeau, Hugo Herbelin and Jean-Marc Notin maintained coqdoc. Frédéric Besson maintained the Micromega platform for deciding systems of inequalities. Pierre Courtieu maintained the support for the Proof General Emacs interface. Claude Marché maintained the plugin for calling external provers (dp). Yves Bertot made some improvements to the libraries of lists and integers. Matthias Puech improved the search functions. Guillaume Melquiond usefully contributed here and there. Yann Régis-Gianas grounded the support for Unicode on a more standard and more robust basis.

Though invisible from outside, Arnaud Spiwack improved the general process of management of existential variables. Pierre Letouzey and Stéphane Glondu improved the compilation scheme of the Coq archive. Vincent Gross provided support to CoqIDE. Jean-Marc Notin provided support for benchmarking and archiving.

Many users helped by reporting problems, providing patches, suggesting improvements or making useful comments, either on the bug tracker or on the Coq-Club mailing list. This includes but not exhaustively Cédric Auger, Arthur Charguéraud, François Garillot, Georges Gonthier, Robin Green, Stéphane Lescuyer, Eelis van der Weegen, ...

Though not directly related to the implementation, special thanks are going to Yves Bertot, Pierre Castéran, Adam Chlipala, and Benjamin Pierce for the excellent teaching materials they provided.

Paris, April 2010
Hugo Herbelin

3.9.2 Details of changes

Rewriting tactics

- Tactic ”rewrite” now supports rewriting on ad hoc equalities such as eq_true.
- ”Hint Rewrite” now checks that the lemma looks like an equation.
- New tactic ”etransitivity”.
- Support for heterogeneous equality (JMeq) in ”injection” and ”discriminate”.
- Tactic ”subst” now supports heterogeneous equality and equality proofs that are dependent (use ”simple subst” for preserving compatibility).
- Added support for Leibniz-rewriting of dependent hypotheses.
- Renamed ”Morphism” into ”Proper” and ”respect” into ”proper_prf” (possible source of incompatibility). A partial fix is to define ”Notation Morphism R f := (Proper (R%signature) f).”
- New tactic variants ”rewrite* by” and ”autorewrite*” that rewrite respectively the first and all matches whose side-conditions are solved.
- ”Require Import Setoid” does not export all of ”Morphisms” and ”RelationClasses” anymore (possible source of incompatibility, fixed by importing ”Morphisms” too).
- Support added for using Chung-Kil Hur’s Heq library for rewriting over heterogeneous equality (courtesy of the library’s author).
- Tactic ”replace” supports matching terms with holes.

Automation tactics

- Tactic `intuition` now preserves inner `iff` and `not` (exceptional source of incompatibilities solvable by redefining `intuition` as `unfold iff, not in *; intuition`, or, for iff only, by using `Set Intuition Iff Unfolding`.)
- Tactic `tauto` now proves classical tautologies as soon as classical logic (i.e. library `Classical_Prop` or `Classical`) is loaded.
- Tactic `gappa` has been removed from the Dp plugin.
- Tactic `firstorder` now supports the combination of its `using` and `with` options.
- New `Hint Resolve ->` (or `<-`) for declaring iff’s as oriented hints (wish #2104).
- An inductive type as argument of the `using` option of `auto` / `eauto` / `firstorder` is interpreted as using the collection of its constructors.
- New decision tactic ”nsatz” to prove polynomial equations by computation of Groebner bases.

Other tactics

- Tactic ”discriminate” now performs intros before trying to discriminate an hypothesis of the goal (previously it applied intro only if the goal had the form t1<>t2) (exceptional source of incompatibilities - former behavior can be obtained by ”Unset Discriminate Introduction”).
- Tactic ”quote” now supports quotation of arbitrary terms (not just the goal).
- Tactic ”idtac” now displays its ”list” arguments.
- New introduction patterns ”*” for introducing the next block of dependent variables and ”**” for introducing all quantified variables and hypotheses.
- Pattern Unification for existential variables activated in tactics and new option ”Unset Tactic Evars Pattern Unification” to deactivate it.

- Resolution of canonical structure is now part of the tactic's unification algorithm.
- New tactic ”decide lemma with hyp” for rewriting decidability lemmas when one knows which side is true.
- Improved support of dependent goals over objects in dependent types for ”destruct” (rare source of incompatibility that can be avoided by unsetting option ”Dependent Propositions Elimination”).
- Tactic ”exists”, ”eexists”, ”destruct” and ”edestruct” supports iteration using comma-separated arguments.
- Tactic names ”case” and ”elim” now support clauses ”as” and ”in” and become then synonymous of ”destruct” and ”induction” respectively.
- A new tactic name ”exfalso” for the use of 'ex-falso quodlibet' principle. This tactic is simply a shortcut for ”elimtype False”.
- Made quantified hypotheses get the name they would have if introduced in the context (possible but rare source of incompatibilities).
- When applying a component of a conjunctive lemma, ”apply in” (and sequences of ”apply in”) now leave the side conditions of the lemmas uniformly after the main goal (possible source of rare incompatibilities).
- In ”simpl c” and ”change c with d”, c can be a pattern.
- Tactic ”revert” now preserves let-in's making it the exact inverse of ”intro”.
- New tactics ”clear dependent H” and ”revert dependent H” that clears (resp. reverts) H and all the hypotheses that depend on H.
- Ltac's pattern-matching now supports matching metavariables that depend on variables bound upwards in the pattern.

Tactic definitions

- Ltac definitions support Local option for non-export outside modules.
- Support for parsing non-empty lists with separators in tactic notations.
- New command ”Locate Ltac” to get the full name of an Ltac definition.

Notations

- Record syntax `{|x=...; y=...|}` now works inside patterns too.
- Abbreviations from non-imported module now invisible at printing time.
- Abbreviations now use implicit arguments and arguments scopes for printing.
- Abbreviations to pure names now strictly behave like the name they refer to (make redirections of qualified names easier).
- Abbreviations for applied constant now propagate the implicit arguments and arguments scope of the underlying reference (possible source of incompatibilities generally solvable by changing such abbreviations from e.g. `Notation foo' := (foo x)` to `Notation foo' y := (foo x (y:=y))`).
- The ”where” clause now supports multiple notations per defined object.
- Recursive notations automatically expand one step on the left for better factorization; recursion notations inner separators now ensured being tokens.
- Added ”Reserved Infix” as a specific shortcut of the corresponding ”Reserved Notation”.
- Open/Close Scope command supports Global option in sections.

Specification language

- New support for local binders in the syntax of Record/Structure fields.
- Fixpoint/CoFixpoint now support building part or all of bodies using tactics.
- Binders given before ":" in lemmas and in definitions built by tactics are now automatically introduced (possible source of incompatibility that can be resolved by invoking "Unset Automatic Introduction").
- New support for multiple implicit arguments signatures per reference.

Module system

- Include Type is now deprecated since Include now accept both modules and module types.
- Declare ML Module supports Local option.
- The sharing between non-logical object and the management of the name-space has been improved by the new "Delta-equivalence" on qualified name.
- The include operator has been extended to high-order structures
- Sequences of Include can be abbreviated via new syntax "<+".
- A module (or module type) can be given several "<:" signatures.
- Interactive proofs are now permitted in module type. Functors can hence be declared as Module Type and be used later to type themselves.
- A functor application can be prefixed by a "!" to make it ignore any "Inline" annotation in the type of its argument(s) (for examples of use of the new features, see libraries Structures and Numbers).
- Coercions are now active only when modules are imported (use "Set Automatic Coercions Import" to get the behavior of the previous versions of Coq).

Extraction

- When using (Recursive) Extraction Library, the filenames are directly the Coq ones with new appropriate extensions : we do not force anymore uncapital first letters for Ocaml and capital ones for Haskell.
- The extraction now tries harder to avoid code transformations that can be dangerous for the complexity. In particular many eta-expansions at the top of functions body are now avoided, clever partial applications will likely be preserved, let-ins are almost always kept, etc.
- In the same spirit, auto-inlining is now disabled by default, except for induction principles, since this feature was producing more frequently weird code than clear gain. The previous behavior can be restored via "Set Extraction AutoInline".
- Unicode characters in identifiers are now transformed into ascii strings that are legal in Ocaml and other languages.
- Harsh support of module extraction to Haskell and Scheme: module hierarchy is flattened, module abbreviations and functor applications are expanded, module types and unapplied functors are discarded.
- Less unsupported situations when extracting modules to Ocaml. In particular module parameters might be alpha-renamed if a name clash is detected.
- Extract Inductive is now possible toward non-inductive types (e.g. nat => int)
- Extraction Implicit: this new experimental command allows to mark some arguments of a function or constructor for removed during extraction, even if these arguments don't fit the usual elimination principles of extraction, for instance the length n of a vector.
- Files ExtrOcaml*.v in plugins/extraction try to provide a library of common extraction commands: mapping of basics types toward Ocaml's counterparts, conversions from/to int and big_int, or even

complete mapping of nat,Z,N to int or big_int, or mapping of ascii to char and string to char list (in this case recognition of ascii constants is hard-wired in the extraction).

Program

- Streamlined definitions using well-founded recursion and measures so that they can work on any subset of the arguments directly (uses currying).
- Try to automatically clear structural fixpoint prototypes in obligations to avoid issues with opacity.
- Use return type clause inference in pattern-matching as in the standard typing algorithm.
- Support [Local Obligation Tactic] and [Next Obligation with tactic].
- Use [Show Obligation Tactic] to print the current default tactic.
- [fst] and [snd] have maximal implicit arguments in Program now (possible source of incompatibility).

Type classes

- Declaring axiomatic type class instances in Module Type should be now done via new command ”Declare Instance”, while the syntax ”Instance” now always provides a concrete instance, both in and out of Module Type.
- Use [Existing Class foo] to declare foo as a class a posteriori. [foo] can be an inductive type or a constant definition. No projections or instances are defined.
- Various bug fixes and improvements: support for defined fields, anonymous instances, declarations giving terms, better handling of sections and [Context].

Vernacular commands

- New command ”Timeout <n> <command>.” interprets a command and a timeout interrupts the interpretation after <n> seconds.
- New command ”Compute <expr>.” is a shortcut for ”Eval vm_compute in <expr>”.
- New command ”Fail <command>.” interprets a command and is successful iff the command fails on an error (but not an anomaly). Handy for tests and illustration of wrong commands.
- Most commands referring to constant (e.g. Print or About) now support referring to the constant by a notation string.
- New option ”Boolean Equality Schemes” to make generation of boolean equality automatic for datatypes (together with option ”Decidable Equality Schemes”, this replaces deprecated option ”Equality Scheme”).
- Made support for automatic generation of case analysis schemes available to user (governed by option ”Set Case Analysis Schemes”).
- New command `Global`? `Generalizable` `All | No` `Variable | Variables` *ident** to declare which identifiers are generalizable in “ `{}` ` and “ `()` ` binders.
- New command ”Print Opaque Dependencies” to display opaque constants in addition to all variables, parameters or axioms a theorem or definition relies on.
- New command ”Declare Reduction <id> := <conv_expr>”, allowing to write later ”Eval <id> in …”. This command accepts a Local variant.
- Syntax of Implicit Type now supports more than one block of variables of a given type.
- Command ”Canonical Structure” now warns when it has no effects.
- Commands of the form ”Set X” or ”Unset X” now support ”Local” and ”Global” prefixes.

Library

- Use ”standard” Coq names for the properties of eq and identity (e.g. refl_equal is now eq_refl). Support for compatibility is provided.
- The function Compare_dec.nat_compare is now defined directly, instead of relying on lt_eq_lt_dec. The earlier version is still available under the name nat_compare_alt.
- Lemmas in library Relations and Reals have been homogenized a bit.
- The implicit argument of Logic.eq is now maximally inserted, allowing to simply write ”eq” instead of ”@eq _” in morphism signatures.
- Wrongly named lemmas (Zlt_gt_succ and Zlt_succ_gt) fixed (potential source of incompatibilities)
- List library:
  - Definitions of list, length and app are now in Init/Datatypes. Support for compatibility is provided.
  - Definition of Permutation is now in Sorting/Permtation.v
  - Some other light revisions and extensions (possible source of incompatibilities solvable by qualifying names accordingly).
- In ListSet, set_map has been fixed (source of incompatibilities if used).
- Sorting library:
  - new mergesort of worst-case complexity O(n*ln(n)) made available in Mergesort.v;
  - former notion of permutation up to setoid from Permutation.v is deprecated and moved to PermutSetoid.v;
  - heapsort from Heap.v of worst-case complexity O(n*n) is deprecated;
  - new file Sorted.v for some definitions of being sorted.
- Structure library. This new library is meant to contain generic structures such as types with equalities or orders, either in Module version (for now) or Type Classes (still to do):
  - DecidableType.v and OrderedType.v: initial notions for FSets/FMaps, left for compatibility but considered as deprecated.
  - Equalities.v and Orders.v: evolutions of the previous files, with fine-grain Module architecture, many variants, use of Equivalence and other relevant Type Classes notions.
  - OrdersTac.v: a generic tactic for solving chains of (in)equalities over variables. See {Nat,N,Z,P}OrderedType.v for concrete instances.
  - GenericMinMax.v: any ordered type can be equipped with min and max. We derived here all the generic properties of these functions.
- MSets library: an important evolution of the FSets library. ”MSets” stands for Modular (Finite) Sets, by contrast with a forthcoming library of Class (Finite) Sets contributed by S. Lescuyer which will be integrated with the next release of Coq. The main features of MSets are:
  - The use of Equivalence, Proper and other Type Classes features easing the handling of setoid equalities.
  - The interfaces are now stated in iff-style. Old specifications are now derived properties.
  - The compare functions are now pure, and return a ”comparison” value. Thanks to the CompSpec inductive type, reasoning on them remains easy.
  - Sets structures requiring invariants (i.e. sorted lists) are built first as ”Raw” sets (pure objects and separate proofs) and attached with their proofs thanks to a generic functor. ”Raw” sets have now a proper interface and can be manipulated directly.

Note: No Maps yet in MSets. The FSets library is still provided for compatibility, but will probably be considered as deprecated in the next release of Coq.

- Numbers library:
  - The abstract layer (NatInt, Natural/Abstract, Integer/Abstract) has been simplified and enhance thanks to new features of the module system such as Include (see above). It has been extended to Euclidean division (three flavors for integers: Trunc, Floor and Math).
  - The arbitrary-large efficient numbers (BigN, BigZ, BigQ) has also been reworked. They benefit from the abstract layer improvements (especially for div and mod). Note that some specifications have slightly changed (compare, div, mod, shift{r,l}). Ring/Field should work better (true recognition of constants).

Tools

- Option -R now supports binding Coq root read-only.
- New coqtop/coqc option -beautify to reformat .v files (usable e.g. to globally update notations).
- New tool beautify-archive to beautify a full archive of developments.
- New coqtop/coqc option -compat X.Y to simulate the general behavior of previous versions of Coq (provides e.g. support for 8.2 compatibility).

Coqdoc

- List have been revamped. List depth and scope is now determined by an ”offside” whitespace rule.
- Text may be italicized by placing it in _underscores_.
- The ”–index <string>” flag changes the filename of the index.
- The ”–toc-depth <int>” flag limits the depth of headers which are included in the table of contents.
- The ”–lib-name <string>” flag prints ”<string> Foo” instead of ”Library Foo” where library titles are called for. The ”–no-lib-name” flag eliminates the extra title.
- New option ”–parse-comments” to allow parsing of regular `(* *)` comments.
- New option ”–plain-comments” to disable interpretation inside comments.
- New option ”–interpolate” to try and typeset identifiers in Coq escapings using the available globalization information.
- New option ”–external url root” to refer to external libraries.
- Links to section variables and notations now supported.

Internal infrastructure

- To avoid confusion with the repository of user’s contributions, the subdirectory ”contrib” has been renamed into ”plugins”. On platforms supporting ocaml native dynlink, code located there is built as loadable plugins for coqtop.
- An experimental build mechanism via ocamlbuild is provided. From the top of the archive, run ./configure as usual, and then ./build. Feedback about this build mechanism is most welcome. Compiling Coq on platforms such as Windows might be simpler this way, but this remains to be tested.
- The Makefile system has been simplified and factorized with the ocamlbuild system. In particular ”make” takes advantage of .mllib files for building .cma/.cmxa. The .vo files to compile are now listed in several vo.itarget files.

3.10 Version 8.2

3.10.1 Summary of changes

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

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

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

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

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

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

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

Matthieu Sozeau extended the Russell language, ending in an convenient way to write programs of given specifications, Pierre Corbineau extended the Mathematical Proof Language and the automation tools that accompany it, Pierre Letouzey supervised and extended various parts of the standard library, Stéphane Glondu contributed a few tactics and improvements, Jean-Marc Notin provided help in debugging, general maintenance and coqdoc support, Vincent Siles contributed extensions of the Scheme command and of injection.

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

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

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

Palaiseau, June 2008
Hugo Herbelin

3.10.2 Details of changes

Language

- If a fixpoint is not written with an explicit { struct ... }, then all arguments are tried successively (from left to right) until one is found that satisfies the structural decreasing condition.
- New experimental typeclass system giving ad-hoc polymorphism and overloading based on dependent records and implicit arguments.
- New syntax "let 'pat := b in c" for let-binding using irrefutable patterns.
- New syntax "forall {A}, T" for specifying maximally inserted implicit arguments in terms.
- Sort of Record/Structure, Inductive and CoInductive defaults to Type if omitted.
- (Co)Inductive types can be defined as records (e.g. "CoInductive stream := { hd : nat; tl : stream }.")
- New syntax "Theorem id1:t1 ... with idn:tn" for proving mutually dependent statements.
- Support for sort-polymorphism on constants denoting inductive types.
- Several evolutions of the module system (handling of module aliases, functorial module types, an Include feature, etc).
- Prop now a subtype of Set (predicative and impredicative forms).
- Recursive inductive types in Prop with a single constructor of which all arguments are in Prop is now considered to be a singleton type. It consequently supports all eliminations to Prop, Set and Type. As a consequence, Acc_rect has now a more direct proof [possible source of easily fixed incompatibility in case of manual definition of a recursor in a recursive singleton inductive type].

Vernacular commands

- Added option Global to "Arguments Scope" for section surviving.
- Added option "Unset Elimination Schemes" to deactivate the automatic generation of elimination schemes.
- Modification of the Scheme command so you can ask for the name to be automatically computed (e.g. Scheme Induction for nat Sort Set).
- New command "Combined Scheme" to build combined mutual induction principles from existing mutual induction principles.
- New command "Scheme Equality" to build a decidable (boolean) equality for simple inductive datatypes and a decision property over this equality (e.g. Scheme Equality for nat).

- Added option ”Set Equality Scheme” to make automatic the declaration of the boolean equality when possible.
- Source of universe inconsistencies now printed when option ”Set Printing Universes” is activated.
- New option ”Set Printing Existential Instances” for making the display of existential variable instances explicit.
- Support for option ”[id1 ... idn]”, and ”-[id1 ... idn]”, for the ”compute”/”cbv” reduction strategy, respectively meaning reduce only, or everything but, the constants id1 ... idn. ”lazy” alone or followed by ”[id1 ... idn]”, and ”-[id1 ... idn]” also supported, meaning apply all of beta-iota-zeta-delta, possibly restricting delta.
- New command ”Strategy” to control the expansion of constants during conversion tests. It generalizes commands Opaque and Transparent by introducing a range of levels. Lower levels are assigned to constants that should be expanded first.
- New options Global and Local to Opaque and Transparent.
- New command ”Print Assumptions” to display all variables, parameters or axioms a theorem or definition relies on.
- ”Add Rec LoadPath” now provides references to libraries using partially qualified names (this holds also for coqtop/coqc option -R).
- SearchAbout supports negated search criteria, reference to logical objects by their notation, and more generally search of subterms.
- ”Declare ML Module” now allows to import .cmxs files when Coq is compiled in native code with a version of OCaml that supports native Dynlink (>= 3.11).
- Specific sort constraints on Record now taken into account.
- ”Print LoadPath” supports a path argument to filter the display.

Libraries

- Several parts of the libraries are now in Type, in particular FSets, SetoidList, ListSet, Sorting, Zmisc. This may induce a few incompatibilities. In case of trouble while fixing existing development, it may help to simply declare Set as an alias for Type (see file SetIsType).
- New arithmetical library in theories/Numbers. It contains:
  - an abstract modular development of natural and integer arithmetics in Numbers/Natural/Abstract and Numbers/Integer/Abstract
  - an implementation of efficient computational bounded and unbounded integers that can be mapped to processor native arithmetics. See Numbers/Cyclic/Int31 for 31-bit integers and Numbers/Natural/BigN for unbounded natural numbers and Numbers/Integer/BigZ for unbounded integers.
  - some proofs that both older libraries Arith, ZArith and NArith and newer BigN and BigZ implement the abstract modular development. This allows in particular BigN and BigZ to already come with a large database of basic lemmas and some generic tactics (ring),

  This library has still an experimental status, as well as the processor-acceleration mechanism, but both its abstract and its concrete parts are already quite usable and could challenge the use of nat, N and Z in actual developments. Moreover, an extension of this framework to rational numbers is ongoing, and an efficient Q structure is already provided (see Numbers/Rational/BigQ), but this part is currently incomplete (no abstract layer and generic lemmas).
- Many changes in FSets/FMaps. In practice, compatibility with earlier version should be fairly good, but some adaptations may be required.

  - Interfaces of unordered (”weak”) and ordered sets have been factorized thanks to new features of Coq modules (in particular Include), see FSetInterface. Same for maps. Hints in these interfaces have been reworked (they are now placed in a ”set” database).
  - To allow full subtyping between weak and ordered sets, a field ”eq_dec” has been added to OrderedType. The old version of OrderedType is now called MiniOrderedType and functor MOT_to_OT allow to convert to the new version. The interfaces and implementations of sets now contain also such a ”eq_dec” field.
  - FSetDecide, contributed by Aaron Bohannon, contains a decision procedure allowing to solve basic set-related goals (for instance, is a point in a particular set ?). See FSetProperties for examples.
  - Functors of properties have been improved, especially the ones about maps, that now propose some induction principles. Some properties of fold need less hypothesis.
  - More uniformity in implementations of sets and maps: they all use implicit arguments, and no longer export unnecessary scopes (see bug #1347)
  - Internal parts of the implementations based on AVL have evolved a lot. The main files FSetAVL and FMapAVL are now much more lightweight now. In particular, minor changes in some functions has allowed to fully separate the proofs of operational correctness from the proofs of well-balancing: well-balancing is critical for efficiency, but not anymore for proving that these trees implement our interfaces, hence we have moved these proofs into appendix files FSetFullAVL and FMapFullAVL. Moreover, a few functions like union and compare have been modified in order to be structural yet efficient. The appendix files also contains alternative versions of these few functions, much closer to the initial Ocaml code and written via the Function framework.

- Library IntMap, subsumed by FSets/FMaps, has been removed from Coq Standard Library and moved into a user contribution Cachan/IntMap
- Better computational behavior of some constants (eq_nat_dec and le_lt_dec more efficient, Z_lt_le_dec and Positive_as_OT.compare transparent, ...) (exceptional source of incompatibilities).
- Boolean operators moved from module Bool to module Datatypes (may need to rename qualified references in script and force notations || and && to be at levels 50 and 40 respectively).
- The constructors xI and xO of type positive now have postfix notations ”~1” and ”~0”, allowing to write numbers in binary form easily, for instance 6 is 1~1~0 and 4*p is p~0~0 (see BinPos.v).
- Improvements to NArith (Nminus, Nmin, Nmax), and to QArith (in particular a better power function).
- Changes in ZArith: several additional lemmas (used in theories/Numbers), especially in Zdiv, Znumtheory, Zpower. Moreover, many results in Zdiv have been generalized: the divisor may simply be non-null instead of strictly positive (see lemmas with name ending by ”_full”). An alternative file ZOdiv proposes a different behavior (the one of Ocaml) when dividing by negative numbers.
- Changes in Arith: EqNat and Wf_nat now exported from Arith, some constructions on nat that were outside Arith are now in (e.g. iter_nat).
- In SetoidList, eqlistA now expresses that two lists have similar elements at the same position, while the predicate previously called eqlistA is now equivlistA (this one only states that the lists contain the same elements, nothing more).
- Changes in Reals:
  - Most statement in ”sigT” (including the completeness axiom) are now in ”sig” (in case of incompatibility, use proj1_sig instead of projT1, sig instead of sigT, etc).
  - More uniform naming scheme (identifiers in French moved to English, consistent use of 0 – zero – instead of O – letter O –, etc).
  - Lemma on prod_f_SO is now on prod_f_R0.

  - Useless hypothesis of ln_exists1 dropped.
  - New Rlogic.v states a few logical properties about R axioms.
  - RIneq.v extended and made cleaner.

- Slight restructuration of the Logic library regarding choice and classical logic. Addition of files providing intuitionistic axiomatizations of descriptions: Epsilon.v, Description.v and IndefiniteDescription.v.
- Definition of pred and minus made compatible with the structural decreasing criterion for use in fixpoints.
- Files Relations/Rstar.v and Relations/Newman.v moved out to the user contribution repository (contribution CoC_History). New lemmas about transitive closure added and some bound variables renamed (exceptional risk of incompatibilities).
- Syntax for binders in terms (e.g. for ”exists”) supports anonymous names.

Notations, coercions, implicit arguments and type inference

- More automation in the inference of the return clause of dependent pattern-matching problems.
- Experimental allowance for omission of the clauses easily detectable as impossible in pattern-matching problems.
- Improved inference of implicit arguments.
- New options ”Set Maximal Implicit Insertion”, ”Set Reversible Pattern Implicit”, ”Set Strongly Strict Implicit” and ”Set Printing Implicit Defensive” for controlling inference and use of implicit arguments.
- New modifier in ”Implicit Arguments” to force an implicit argument to be maximally inserted.
- New modifier of ”Implicit Arguments” to enrich the set of implicit arguments.
- New options Global and Local to ”Implicit Arguments” for section surviving or non export outside module.
- Level ”constr” moved from 9 to 8.
- Structure/Record now printed as Record (unless option Printing All is set).
- Support for parametric notations defining constants.
- Insertion of coercions below product types refrains to unfold constants (possible source of incompatibility).
- New support for fix/cofix in notations.

Tactic Language

- Second-order pattern-matching now working in Ltac ”match” clauses (syntax for second-order unification variable is ”@?X”).
- Support for matching on let bindings in match context using syntax ”H := body” or ”H := body : type”.
- Ltac accepts integer arguments (syntax is ”ltac:nnn” for nnn an integer).
- The general sequence tactical ”expr_0 ; [expr_1 | ... | expr_n]” is extended so that at most one expr_i may have the form ”expr ..” or just ”..”. Also, n can be different from the number of subgoals generated by expr_0. In this case, the value of expr (or idtac in case of just ”..”) is applied to the intermediate subgoals to make the number of tactics equal to the number of subgoals.
- A name used as the name of the parameter of a lemma (like f in ”apply f_equal with (f:=t)”) is now interpreted as a ltac variable if such a variable exists (this is a possible source of incompatibility and it

can be fixed by renaming the variables of a ltac function into names that do not clash with the lemmas parameter names used in the tactic).

- New syntax ”Ltac tac ::= ...” to rebind a tactic to a new expression.
- ”let rec ... in ... ” now supported for expressions without explicit parameters; interpretation is lazy to the contrary of ”let ... in ...”; hence, the ”rec” keyword can be used to turn the argument of a ”let ... in ...” into a lazy one.
- Patterns for hypotheses types in ”match goal” are now interpreted in type_scope.
- A bound variable whose name is not used elsewhere now serves as metavariable in ”match” and it gets instantiated by an identifier (allow e.g. to extract the name of a statement like ”exists x, P x”).
- New printing of Ltac call trace for better debugging.

Tactics

- New tactics ”apply -> term”, ”apply <- term”, ”apply -> term in ident”, ”apply <- term in ident” for applying equivalences (iff).
- Slight improvement of the hnf and simpl tactics when applied on expressions with explicit occurrences of match or fix.
- New tactics ”eapply in”, ”erewrite”, ”erewrite in”.
- New tactics ”ediscriminate”, ”einjection”, ”esimplify_eq”.
- Tactics ”discriminate”, ”injection”, ”simplify_eq” now support any term as argument. Clause ”with” is also supported.
- Unfoldable references can be given by notation’s string rather than by name in unfold.
- The ”with” arguments are now typed using informations from the current goal: allows support for coercions and more inference of implicit arguments.
- Application of ”f_equal”-style lemmas works better.
- Tactics elim, case, destruct and induction now support variants eelim, ecase, edestruct and einduction.
- Tactics destruct and induction now support the ”with” option and the ”in” clause option. If the option ”in” is used, an equality is added to remember the term to which the induction or case analysis applied (possible source of parsing incompatibilities when destruct or induction is part of a let-in expression in Ltac; extra parentheses are then required).
- New support for ”as” clause in tactics ”apply in” and ”eapply in”.
- Some new intro patterns:
  - intro pattern ”?A” genererates a fresh name based on A. Caveat about a slight loss of compatibility: Some intro patterns don’t need space between them. In particular intros ?a?b used to be legal and equivalent to intros ? a ? b. Now it is still legal but equivalent to intros ?a ?b.
  - intro pattern ”(A & ... & Y & Z)” synonym to ”(A,....,(Y,Z)))))” for right-associative constructs like /or exists.
- Several syntax extensions concerning ”rewrite”:
  - ”rewrite A,B,C” can be used to rewrite A, then B, then C. These rewrites occur only on the first subgoal: in particular, side-conditions of the ”rewrite A” are not concerned by the ”rewrite B,C”.
  - ”rewrite A by tac” allows to apply tac on all side-conditions generated by the ”rewrite A”.
  - ”rewrite A at n” allows to select occurrences to rewrite: rewrite only happen at the n-th exact occurrence of the first successful matching of A in the goal.

  - ”rewrite 3 A” or ”rewrite 3!A” is equivalent to ”rewrite A,A,A”.
  - ”rewrite !A” means rewriting A as long as possible (and at least once).
  - ”rewrite 3?A” means rewriting A at most three times.
  - ”rewrite ?A” means rewriting A as long as possible (possibly never).
  - many of the above extensions can be combined with each other.
- Introduction patterns better respect the structure of context in presence of missing or extra names in nested disjunction-conjunction patterns [possible source of rare incompatibilities].
- New syntax ”rename a into b, c into d” for ”rename a into b; rename c into d”
- New tactics ”dependent induction/destruction H [generalizing id_1 .. id_n]” to do induction-inversion on instantiated inductive families à la BasicElim.
- Tactics ”apply” and ”apply in” now able to reason modulo unfolding of constants (possible source of incompatibility in situations where apply may fail, e.g. as argument of a try or a repeat and in a ltac function); versions that do not unfold are renamed into ”simple apply” and ”simple apply in” (usable for compatibility or for automation).
- Tactics ”apply” and ”apply in” now able to traverse conjunctions and to select the first matching lemma among the components of the conjunction; tactic ”apply” also able to apply lemmas of conclusion an empty type.
- Tactic ”apply” now supports application of several lemmas in a row.
- Tactics ”set” and ”pose” can set functions using notation ”(f x1..xn := c)”.
- New tactic ”instantiate” (without argument).
- Tactic firstorder ”with” and ”using” options have their meaning swapped for consistency with auto/eauto (source of incompatibility).
- Tactic ”generalize” now supports ”at” options to specify occurrences and ”as” options to name the quantified hypotheses.
- New tactic ”specialize H with a” or ”specialize (H a)” allows to transform in-place a universally-quantified hypothesis (H : forall x, T x) into its instantiated form (H : T a). Nota: ”specialize” was in fact there in earlier versions of Coq, but was undocumented, and had a slightly different behavior.
- New tactic ”contradict H” can be used to solve any kind of goal as long as the user can provide afterwards a proof of the negation of the hypothesis H. If H is already a negation, say ~T, then a proof of T is asked. If the current goal is a negation, say ~U, then U is saved in H afterwards, hence this new tactic ”contradict” extends earlier tactic ”swap”, which is now obsolete.
- Tactics f_equal is now done in ML instead of Ltac: it now works on any equality of functions, regardless of the arity of the function.
- New options ”before id”, ”at top”, ”at bottom” for tactics ”move”/”intro”.
- Some more debug of reflexive omega (`romega`), and internal clarifications. Moreover, romega now has a variant `romega with *` that can be also used on non-Z goals (nat, N, positive) via a call to a translation tactic named zify (its purpose is to Z-ify your goal...). This zify may also be used independently of romega.
- Tactic ”remember” now supports an ”in” clause to remember only selected occurrences of a term.
- Tactic ”pose proof” supports name overwriting in case of specialization of an hypothesis.
- Semi-decision tactic ”jp” for first-order intuitionistic logic moved to user contributions (subsumed by ”firstorder”).

Program

- Moved useful tactics in theories/Program and documented them.
- Add Program.Basics which contains standard definitions for functional programming (id, apply, flip...)
- More robust obligation handling, dependent pattern-matching and well-founded definitions.
- New syntax " dest term as pat in term " for destructing objects using an irrefutable pattern while keeping equalities (use this instead of "let" in Programs).
- Program CoFixpoint is accepted, Program Fixpoint uses the new way to infer which argument decreases structurally.
- Program Lemma, Axiom etc... now permit to have obligations in the statement iff they can be automatically solved by the default tactic.
- Renamed "Obligations Tactic" command to "Obligation Tactic".
- New command "Preterm [of id]" to see the actual term fed to Coq for debugging purposes.
- New option "Transparent Obligations" to control the declaration of obligations as transparent or opaque. All obligations are now transparent by default, otherwise the system declares them opaque if possible.
- Changed the notations "left" and "right" to "in_left" and "in_right" to hide the proofs in standard disjunctions, to avoid breaking existing scripts when importing Program. Also, put them in program_scope.

Type Classes

- New "Class", "Instance" and "Program Instance" commands to define classes and instances documented in the reference manual.
- New binding construct " [Class_1 param_1 .. param_n, Class_2 ...] " for binding type classes, usable everywhere.
- New command " Print Classes " and " Print Instances some_class " to print tables for typeclasses.
- New default eauto hint database "typeclass_instances" used by the default typeclass instance search tactic.
- New theories directory "theories/Classes" for standard typeclasses declarations. Module Classes.RelationClasses is a typeclass port of Relation_Definitions plus a generic development of algebra on n-ary heterogeneous predicates.

Setoid rewriting

- Complete (and still experimental) rewrite of the tactic based on typeclasses. The old interface and semantics are almost entirely respected, except:
  - Import Setoid is now mandatory to be able to call setoid_replace and declare morphisms.
  - "–>", "++>" and "==>" are now right associative notations declared at level 55 in scope signature_scope. Their introduction may break existing scripts that defined them as notations with different levels.
  - One needs to use [Typeclasses unfold [cst]] if [cst] is used as an abbreviation hiding products in types of morphisms, e.g. if ones redefines [relation] and declares morphisms whose type mentions [relation].
  - The [setoid_rewrite]'s semantics change when rewriting with a lemma: it can rewrite two different instantiations of the lemma at once. Use [setoid_rewrite H at 1] for (almost) the usual semantics. [setoid_rewrite] will also try to rewrite under binders now, and can succeed on different terms

than before. In particular, it will unify under let-bound variables. When called through [rewrite], the semantics are unchanged though.

  - [Add Morphism term : id] has different semantics when used with parametric morphism: it will try to find a relation on the parameters too. The behavior has also changed with respect to default relations: the most recently declared Setoid/Relation will be used, the documentation explains how to customize this behavior.
  - Parametric Relation and Morphism are declared differently, using the new [Add Parametric] commands, documented in the manual.
  - Setoid_Theory is now an alias to Equivalence, scripts building objects of type Setoid_Theory need to unfold (or ”red”) the definitions of Reflexive, Symmetric and Transitive in order to get the same goals as before. Scripts which introduced variables explicitly will not break.
  - The order of subgoals when doing [setoid_rewrite] with side-conditions is always the same: first the new goal, then the conditions.

- New standard library modules `Classes.Morphisms` declares standard morphisms on `refl` / `sym` / `trans` relations. `Classes.Morphisms_Prop` declares morphisms on propositional connectives and `Classes.Morphisms_Relations` on generalized predicate connectives. `Classes.Equivalence` declares notations and tactics related to equivalences and `Classes.SetoidTactics` defines the setoid_replace tactics and some support for the `Add *` interface, notably the tactic applied automatically before each `Add Morphism` proof.
- User-defined subrelations are supported, as well as higher-order morphisms and rewriting under binders. The tactic is also extensible entirely in Ltac. The documentation has been updated to cover these features.
- [setoid_rewrite] and [rewrite] now support the [at] modifier to select occurrences to rewrite, and both use the [setoid_rewrite] code, even when rewriting with leibniz equality if occurrences are specified.

Extraction

- Improved behavior of the Caml extraction of modules: name clashes should not happen anymore.
- The command Extract Inductive has now a syntax for infix notations. This allows in particular to map Coq lists and pairs onto Caml ones:
  - Extract Inductive list => list [”[]” ”(::)”].
  - Extract Inductive prod => ”(*)” [”(,)”].
- In pattern matchings, a default pattern ”| _ -> ...” is now used whenever possible if several branches are identical. For instance, functions corresponding to decidability of equalities are now linear instead of quadratic.
- A new instruction Extraction Blacklist id1 .. idn allows to prevent filename conflits with existing code, for instance when extracting module List to Ocaml.

CoqIDE

- CoqIDE font defaults to monospace so as indentation to be meaningful.
- CoqIDE supports nested goals and any other kind of declaration in the middle of a proof.
- Undoing non-tactic commands in CoqIDE works faster.
- New CoqIDE menu for activating display of various implicit informations.
- Added the possibility to choose the location of tabs in coqide: (in Edit->Preferences->Misc)
- New Open and Save As dialogs in CoqIDE which filter `*.v` files.

Tools

- New stand-alone .vo files verifier ”coqchk”.
- Extended -I coqtop/coqc option to specify a logical dir: ”-I dir -as coqdir”.
- New coqtop/coqc option -exclude-dir to exclude subdirs for option -R.
- The binary ”parser” has been renamed to ”coq-parser”.
- Improved coqdoc and dump of globalization information to give more meta-information on identifiers. All categories of Coq definitions are supported, which makes typesetting trivial in the generated documentation. Support for hyperlinking and indexing developments in the tex output has been implemented as well.

Miscellaneous

- Coq installation provides enough files so that Ocaml’s extensions need not the Coq sources to be compiled (this assumes O’Caml 3.10 and Camlp5).
- New commands ”Set Whelp Server” and ”Set Whelp Getter” to customize the Whelp search tool.
- Syntax of ”Test Printing Let ref” and ”Test Printing If ref” changed into ”Test Printing Let for ref” and ”Test Printing If for ref”.
- An overhauled build system (new Makefiles); see dev/doc/build-system.txt.
- Add -browser option to configure script.
- Build a shared library for the C part of Coq, and use it by default on non-(Windows or MacOS) systems. Bytecode executables are now pure. The behaviour is configurable with -coqrunbyteflags, -coqtoolsbyteflags and -custom configure options.
- Complexity tests can be skipped by setting the environment variable COQTEST_SKIPCOMPLEXITY.

3.11 Version 8.1

3.11.1 Summary of changes

Coq version 8.1 adds various new functionalities.

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

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

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

Claudio Sacerdoti Coen added new features to the module system.

Benjamin Grégoire, Assia Mahboubi and Bruno Barras developed a new, more efficient and more general simplification algorithm for rings and semirings.

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

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

Hugo Herbelin implemented matching on disjunctive patterns.

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

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

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

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

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

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

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

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

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

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

Palaiseau, July 2006
Hugo Herbelin

3.11.2 Details of changes in 8.1beta

Logic

- Added sort-polymorphism on inductive families
- Allowance for recursively non uniform parameters in inductive types

Syntax

- No more support for version 7 syntax and for translation to version 8 syntax.
- In fixpoints, the { struct ... } annotation is not mandatory any more when only one of the arguments has an inductive type
- Added disjunctive patterns in match-with patterns

- Support for primitive interpretation of string literals
- Extended support for Unicode ranges

Vernacular commands

- Added ”Print Ltac qualid” to print a user defined tactic.
- Added ”Print Rewrite HintDb” to print the content of a DB used by autorewrite.
- Added ”Print Canonical Projections”.
- Added ”Example” as synonym of ”Definition”.
- Added ”Proposition” and ”Corollary” as extra synonyms of ”Lemma”.
- New command ”Whelp” to send requests to the Helm database of proofs formalized in the Calculus of Inductive Constructions.
- Command ”functional induction” has been re-implemented from the new ”Function” command.

Ltac and tactic syntactic extensions

- New primitive ”external” for communication with tool external to Coq
- New semantics for ”match t with”: if a clause returns a tactic, it is now applied to the current goal. If it fails, the next clause or next matching subterm is tried (i.e. it behaves as ”match goal with” does). The keyword ”lazymatch” can be used to delay the evaluation of tactics occurring in matching clauses.
- Hint base names can be parametric in auto and trivial.
- Occurrence values can be parametric in unfold, pattern, etc.
- Added entry constr_may_eval for tactic extensions.
- Low-priority term printer made available in ML-written tactic extensions.
- ”Tactic Notation” extended to allow notations of tacticals.

Tactics

- New implementation and generalization of `setoid_*` (`setoid_rewrite`, `setoid_symmetry`, `setoid_transitivity`, `setoid_reflexivity` and `autorewite`). New syntax for declaring relations and morphisms (old syntax still working with minor modifications, but deprecated).
- New implementation (still experimental) of the ring tactic with a built-in notion of coefficients and a better usage of setoids.
- New conversion tactic ”vm_compute”: evaluates the goal (or an hypothesis) with a call-by-value strategy, using the compiled version of terms.
- When rewriting H where H is not directly a Coq equality, search first H for a registered setoid equality before starting to reduce in H. This is unlikely to break any script. Should this happen nonetheless, one can insert manually some ”unfold ... in H” before rewriting.
- Fixed various bugs about (setoid) rewrite ... in ... (in particular bug #5941)
- ”rewrite ... in” now accepts a clause as place where to rewrite instead of just a simple hypothesis name. For instance: `rewrite H in H1,H2 |- *` means `rewrite H in H1; rewrite H in H2; rewrite H` `rewrite H in * |-` will do try `rewrite H in Hi` for all hypothesis Hi <> H.
- Added ”dependent rewrite term” and ”dependent rewrite term in hyp”.
- Added ”autorewrite with ... in hyp [using ...]”.
- Tactic ”replace” now accepts a ”by” tactic clause.
- Added ”clear - id” to clear all hypotheses except the ones depending in id.

- The argument of Declare Left Step and Declare Right Step is now a term (it used to be a reference).
- Omega now handles arbitrary precision integers.
- Several bug fixes in Reflexive Omega (romega).
- Idtac can now be left implicit in a [...|...] construct: for instance, [foo | | bar] stands for [foo | idtac | bar].
- Fixed a ”fold” bug (non critical but possible source of incompatibilities).
- Added classical_left and classical_right which transforms `|- A \/ B` into `~B |- A` and `~A |- B` respectively.
- Added command ”Declare Implicit Tactic” to set up a default tactic to be used to solve unresolved subterms of term arguments of tactics.
- Better support for coercions to Sortclass in tactics expecting type arguments.
- Tactic ”assert” now accepts ”as” intro patterns and ”by” tactic clauses.
- New tactic ”pose proof” that generalizes ”assert (id:=p)” with intro patterns.
- New introduction pattern ”?” for letting Coq choose a name.
- Introduction patterns now support side hypotheses (e.g. intros [|] on ”(nat -> nat) -> nat” works).
- New introduction patterns ”->” and ”<-” for immediate rewriting of introduced hypotheses.
- Introduction patterns coming after non trivial introduction patterns now force full introduction of the first pattern (e.g. `intros [[|] p]` on `nat->nat->nat` now behaves like `intros [[|?] p]`)
- Added ”eassumption”.
- Added option ’using lemmas’ to auto, trivial and eauto.
- Tactic ”congruence” is now complete for its intended scope (ground equalities and inequalities with constructors). Furthermore, it tries to equates goal and hypotheses.
- New tactic ”rtauto” solves pure propositional logic and gives a reflective version of the available proof.
- Numbering of ”pattern”, ”unfold”, ”simpl”, ... occurrences in ”match with” made consistent with the printing of the return clause after the term to match in the ”match-with” construct (use ”Set Printing All” to see hidden occurrences).
- Generalization of induction ”induction x1...xn using scheme” where scheme is an induction principle with complex predicates (like the ones generated by function induction).
- Some small Ltac tactics has been added to the standard library (file Tactics.v):
  - f_equal : instead of using the different f_equalX lemmas
  - case_eq : a ”case” without loss of information. An equality stating the current situation is generated in every sub-cases.
  - swap : for a negated goal ~B and a negated hypothesis H:~A, swap H asks you to prove A from hypothesis B
  - revert : revert H is generalize H; clear H.

Extraction

- All type parts should now disappear instead of sometimes producing _ (for instance in Map.empty).
- Haskell extraction: types of functions are now printed, better unsafeCoerce mechanism, both for hugs and ghc.
- Scheme extraction improved, see http://www.pps.jussieu.fr/~letouzey/scheme.

- Many bug fixes.

Modules

- Added ”Locate Module qualid” to get the full path of a module.
- Module/Declare Module syntax made more uniform.
- Added syntactic sugar ”Declare Module Export/Import” and ”Module Export/Import”.
- Added syntactic sugar ”Module M(Export/Import X Y: T)” and ”Module Type M(Export/Import X Y: T)” (only for interactive definitions)
- Construct ”with” generalized to module paths: T with (Definition|Module) M1.M2....Mn.l := l'.

Notations

- Option ”format” aware of recursive notations.
- Added insertion of spaces by default in recursive notations w/o separators.
- No more automatic printing box in case of user-provided printing ”format”.
- New notation ”exists! x:A, P” for unique existence.
- Notations for specific numerals now compatible with generic notations of numerals (e.g. ”1” can be used to denote the unit of a group without hiding 1%nat)

Libraries

- New library on String and Ascii characters (contributed by L. Thery).
- New library FSets+FMaps of finite sets and maps.
- New library QArith on rational numbers.
- Small extension of Zmin.V, new Zmax.v, new Zminmax.v.
- Reworking and extension of the files on classical logic and description principles (possible incompatibilities)
- Few other improvements in ZArith potentially exceptionally breaking the compatibility (useless hypothesys of Zgt_square_simpl and Zlt_square_simpl removed; fixed names mentioning letter O instead of digit 0; weaken premises in Z_lt_induction).
- Restructuration of Eqdep_dec.v and Eqdep.v: more lemmas in Type.
- Znumtheory now contains a gcd function that can compute within Coq.
- More lemmas stated on Type in Wf.v, removal of redundant Acc_iter and Acc_iter2.
- Change of the internal names of lemmas in OmegaLemmas.
- Acc in Wf.v and clos_refl_trans in Relation_Operators.v now rely on the allowance for recursively non uniform parameters (possible source of incompatibilities: explicit pattern-matching on these types may require to remove the occurrence associated to their recursively non uniform parameter).
- Coq.List.In_dec has been set transparent (this may exceptionally break proof scripts, set it locally opaque for compatibility).
- More on permutations of lists in List.v and Permutation.v.
- List.v has been much expanded.
- New file SetoidList.v now contains results about lists seen with respect to a setoid equality.
- Library NArith has been expanded, mostly with results coming from Intmap (for instance a bitwise xor), plus also a bridge between N and Bitvector.

- Intmap has been reorganized. In particular its address type ”addr” is now N. User contributions known to use Intmap have been adapted accordingly. If you’re using this library please contact us. A wrapper FMapIntMap now presents Intmap as a particular implementation of FMaps. New developments are strongly encouraged to use either this wrapper or any other implementations of FMap instead of using directly this obsolete Intmap.

Tools

- New semantics for coqtop options (”-batch” expects option ”-top dir” for loading vernac file that contains definitions).
- Tool coq_makefile now removes custom targets that are file names in ”make clean”
- New environment variable COQREMOTEBROWSER to set the command invoked to start the remote browser both in Coq and coqide. Standard syntax: ”%s” is the placeholder for the URL.

3.11.3 Details of changes in 8.1gamma

Syntax

- changed parsing precedence of let/in and fun constructions of Ltac: let x := t in e1; e2 is now parsed as let x := t in (e1;e2).

Language and commands

- Added sort-polymorphism for definitions in Type (but finally abandoned).
- Support for implicit arguments in the types of parameters in (co-)fixpoints and (co-)inductive declarations.
- Improved type inference: use as much of possible general information. before applying irreversible unification heuristics (allow e.g. to infer the predicate in ”(exist _ 0 (refl_equal 0) : {n:nat | n=0 })”).
- Support for Miller-Pfenning’s patterns unification in type synthesis (e.g. can infer P such that P x y = phi(x,y)).
- Support for ”where” clause in cofixpoint definitions.
- New option ”Set Printing Universes” for making Type levels explicit.

Tactics

- Improved implementation of the ring and field tactics. For compatibility reasons, the previous tactics are renamed as legacy ring and legacy field, but should be considered as deprecated.
- New declarative mathematical proof language.
- Support for argument lists of arbitrary length in Tactic Notation.
- `rewrite ... in H` now fails if `H` is used either in an hypothesis or in the goal.
- The semantics of `rewrite ... in *` has been slightly modified (see doc).
- Support for `as` clause in tactic injection.
- New forward-reasoning tactic ”apply in”.
- Ltac fresh operator now builds names from a concatenation of its arguments.
- New ltac tactic ”remember” to abstract over a subterm and keep an equality
- Support for Miller-Pfenning’s patterns unification in apply/rewrite/... (may lead to few incompatibilities - generally now useless tactic calls).

Bug fixes

- Fix for notations involving basic ”match” expressions.
- Numerous other bugs solved (a few fixes may lead to incompatibilities).

3.11.4 Details of changes in 8.1

Bug fixes

- Many bugs have been fixed (cf coq-bugs web page)

Tactics

- New tactics ring, ring_simplify and new tactic field now able to manage power to a positive integer constant. Tactic ring on Z and R, and field on R manage power (may lead to incompatibilities with V8.1gamma).
- Tactic field_simplify now applicable in hypotheses.
- New field_simplify_eq for simplifying field equations into ring equations.
- Tactics ring, ring_simplify, field, field_simplify and field_simplify_eq all able to apply user-given equations to rewrite monoms on the fly (see documentation).

Libraries

- New file ConstructiveEpsilon.v defining an epsilon operator and proving the axiom of choice constructively for a countable domain and a decidable predicate.

3.12 Version 8.0

3.12.1 Summary of changes

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

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

- a more uniform, purified style: all constructions are now lowercase, with a functional programming perfume (e.g. abstraction is now written fun), and more directly accessible to the novice (e.g. dependent product is now written forall and allows omission of types). Also, parentheses 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 restructuring and uniformization 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 with Coq. Close in style to 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 reflection 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 implementer of the notion of interpretation scopes and of the commands for easily adding new notations.

Hugo Herbelin is the main implementer of the restructured standard library.

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

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

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

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

Jean-Christophe Filliâtre, Pierre Letouzey, Hugo Herbelin and other contributors from Sophia-Antipolis and Nijmegen participated in extending the library.

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

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

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

3.12.2 Details of changes in 8.0beta old syntax

Logic

- Set now predicative by default
- New option -impredicative-set to set Set impredicative
- The standard library doesn't need impredicativity of Set and is compatible with the classical axioms which contradict Set impredicativity

Syntax for arithmetic

- Notation ”=” and ”<>” in Z and R are no longer implicitly in Z or R (with possible introduction of a coercion), use <Z>...=... or <Z>...<>... instead
- Locate applied to a simple string (e.g. ”+”) searches for all notations containing this string

Vernacular commands

- ”Declare ML Module” now allows to import .cma files. This avoids to use a bunch of ”Declare ML Module” statements when using several ML files.
- ”Set Printing Width n” added, allows to change the size of width printing.
- ”Implicit Variables Type x,y:t” (new syntax: ”Implicit Types x y:t”) assigns default types for binding variables.
- Declarations of Hints and Notation now accept a ”Local” flag not to be exported outside the current file even if not in section
- ”Print Scopes” prints all notations
- New command ”About name” for light printing of type, implicit arguments, etc.
- New command ”Admitted” to declare incompletely proven statement as axioms
- New keyword ”Conjecture” to declare an axiom intended to be provable
- SearchAbout can now search for lemmas referring to more than one constant and on substrings of the name of the lemma
- ”Print Implicit” displays the implicit arguments of a constant
- Locate now searches for all names having a given suffix
- New command ”Functional Scheme” for building an induction principle from a function defined by case analysis and fix.

Commands

- new coqtop/coqc option -dont-load-proofs not to load opaque proofs in memory

Implicit arguments

- Inductive in sections declared with implicits now ”discharged” with implicits (like constants and variables)
- Implicit Arguments flags are now synchronous with reset
- New switch ”Unset/Set Printing Implicits” (new syntax: ”Unset/Set Printing Implicit”) to globally control printing of implicits

Grammar extensions

- Many newly supported UTF-8 encoded unicode blocks - Greek letters (0380-03FF), Hebrew letters (U05D0-05EF), letter-like symbols (2100-214F, that includes double N,Z,Q,R), prime signs (from 2080-2089) and characters from many written languages are valid in identifiers - mathematical operators (2200-22FF), supplemental mathematical operators (2A00-2AFF), miscellaneous technical (2300-23FF that includes sqrt symbol), miscellaneous symbols (2600-26FF), arrows (2190-21FF and 2900-297F), invisible mathematical operators (from 2080-2089), ... are valid symbols

Library

- New file about the factorial function in Arith
- An additional elimination Acc_iter for Acc, simpler than Acc_rect. This new elimination principle is used for definition well_founded_induction.

- New library NArith on binary natural numbers
- R is now of type Set
- Restructuration in ZArith library
  - ”true_sub” used in Zplus now a definition, not a local one (source of incompatibilities in proof referring to true_sub, may need extra Unfold)
  - Some lemmas about minus moved from fast_integer to Arith/Minus.v (le_minus, lt_mult_left) (theoretical source of incompatibilities)
  - Several lemmas moved from auxiliary.v and zarith_aux.v to fast_integer.v (theoretical source of incompatibilities)
  - Variables names of iff_trans changed (source of incompatibilities)
  - ZArith lemmas named `OMEGA` something or `fast_` something, and lemma `new_var` are now out of ZArith (except `OMEGA2`)
  - Redundant ZArith lemmas have been renamed: for the following pairs, use the second name (Zle_Zmult_right2, Zle_mult_simpl), (OMEGA2, Zle_0_plus), (Zplus_assoc_l, Zplus_assoc), (Zmult_one, Zmult_1_n), (Zmult_assoc_l, Zmult_assoc), (Zmult_minus_distr, Zmult_Zminus_distr_l) (add_un_double_moins_un_xO, is_double_moins_un), (Rlt_monotony_rev,Rlt_monotony_contra) (source of incompatibilities)
- Few minor changes (no more implicit arguments in Zmult_Zminus_distr_l and Zmult_Zminus_distr_r, lemmas moved from Zcomplements to other files) (rare source of incompatibilities)
- New lemmas provided by users added

Tactic language

- Fail tactic now accepts a failure message
- Idtac tactic now accepts a message
- New primitive tactic ”FreshId” (new syntax: ”fresh”) to generate new names
- Debugger prints levels of calls

Tactics

- Replace can now replace proofs also
- Fail levels are now decremented at ”Match Context” blocks only and if the right-hand-side of ”Match term With” are tactics, these tactics are never evaluated immediately and do not induce backtracking (in contrast with ”Match Context”)
- Quantified names now avoid global names of the current module (like Intro names did) [source of rare incompatibilities: 2 changes in the set of user contribs]
- NewDestruct/NewInduction accepts intro patterns as introduction names
- NewDestruct/NewInduction now work for non-inductive type using option ”using”
- A NewInduction naming bug for inductive types with functional arguments (e.g. the accessibility predicate) has been fixed (source of incompatibilities)
- Symmetry now applies to hypotheses too
- Inversion now accept option ”as [...]” to name the hypotheses
- Contradiction now looks also for contradictory hypotheses stating ~A and A (source of incompatibility)
- ”Contradiction c” try to find an hypothesis in context which contradicts the type of c

- Ring applies to new library NArith (require file NArithRing)
- Field now works on types in Set
- Auto with reals now try to replace le by ge (Rge_le is no longer an immediate hint), resulting in shorter proofs
- Instantiate now works in hyps (syntax : Instantiate in ...)
- Some new tactics : EConstructor, ELeft, Eright, ESplit, EExists
- New tactic ”functional induction” to perform case analysis and induction following the definition of a function.
- Clear now fails when trying to remove a local definition used by a constant appearing in the current goal

Extraction (See details in plugins/extraction/CHANGES)

- The old commands: (Recursive) Extraction Module M. are now: (Recursive) Extraction Library M. To use these commands, M should come from a library M.v
- The other syntax Extraction & Recursive Extraction now accept module names as arguments.

Bugs

- see coq-bugs server for the complete list of fixed bugs

Miscellaneous

- Implicit parameters of inductive types definition now taken into account for inferring other implicit arguments

Incompatibilities

- Persistence of true_sub (4 incompatibilities in Coq user contributions)
- Variable names of some constants changed for a better uniformity (2 changes in Coq user contributions)
- Naming of quantified names in goal now avoid global names (2 occurrences)
- NewInduction naming for inductive types with functional arguments (no incompatibility in Coq user contributions)
- Contradiction now solve more goals (source of 2 incompatibilities)
- Merge of eq and eqT may exceptionally result in subgoals now solved automatically
- Redundant pairs of ZArith lemmas may have different names: it may cause ”Apply/Rewrite with” to fail if using the first name of a pair of redundant lemmas (this is solved by renaming the variables bound by ”with”; 3 incompatibilities in Coq user contribs)
- ML programs referring to constants from fast_integer.v must use ”Coqlib.gen_constant_modules Coqlib.zarith_base_modules” instead

3.12.3 Details of changes in 8.0beta new syntax

New concrete syntax

- A completely new syntax for terms
- A more uniform syntax for tactics and the tactic language
- A few syntactic changes for vernacular commands
- A smart automatic translator translating V8.0 files in old syntax to files valid for V8.0

Syntax extensions

- ”Grammar” for terms disappears
- ”Grammar” for tactics becomes ”Tactic Notation”
- ”Syntax” disappears
- Introduction of a notion of interpretation scope allowing to use the same notations in various contexts without using specific delimiters (e.g the same expression ”4<=3+x” is interpreted either in ”nat”, ”positive”, ”N” (previously ”entier”), ”Z”, ”R”, depending on which interpretation scope is currently open) [see documentation for details]
- Notation now mandatorily requires a precedence and associativity (default was to set precedence to 1 and associativity to none)

Revision of the standard library

- Many lemmas and definitions names have been made more uniform mostly in Arith, NArith, ZArith and Reals (e.g : ”times” -> ”Pmult”, ”times_sym” -> ”Pmult_comm”, ”Zle_Zmult_pos_right” -> ”Zmult_le_compat_r”, ”SUPERIEUR” -> ”Gt”, ”ZERO” -> ”Z0”)
- Order and names of arguments of basic lemmas on nat, Z, positive and R have been made uniform.
- Notions of Coq initial state are declared with (strict) implicit arguments
- eq merged with eqT: old eq disappear, new eq (written =) is old eqT and new eqT is syntactic sugar for new eq (notation == is an alias for = and is written as it, exceptional source of incompatibilities)
- Similarly, ex, ex2, all, identity are merged with exT, exT2, allT, identityT
- Arithmetical notations for nat, positive, N, Z, R, without needing any backquote or double-backquotes delimiters.
- In Lists: new concrete notations; argument of nil is now implicit
- All changes in the library are taken in charge by the translator

Semantical changes during translation

- Recursive keyword set by default (and no longer needed) in Tactic Definition
- Set Implicit Arguments is strict by default in new syntax
- reductions in hypotheses of the form ”... in H” now apply to the type also if H is a local definition
- etc

Gallina

- New syntax of the form ”Inductive bool : Set := true, false : bool.” for enumerated types
- Experimental syntax of the form p.(fst) for record projections (activable with option ”Set Printing Projections” which is recognized by the translator)

Known problems of the automatic translation

- iso-latin-1 characters are no longer supported: move your files to 7-bits ASCII or unicode before translation (switch to unicode is automatically done if a file is loaded and saved again by coqide)
- Renaming in ZArith: incompatibilities in Coq user contribs due to merging names INZ, from Reals, and inject_nat.
- Renaming and new lemmas in ZArith: may clash with names used by users
- Restructuration of ZArith: replace requirement of specific modules in ZArith by ”Require Import ZArith_base” or ”Require Import ZArith”

- Some implicit arguments must be made explicit before translation: typically for ”length nil”, the implicit argument of length must be made explicit
- Grammar rules, Infix notations and V7.4 Notations must be updated wrt the new scheme for syntactic extensions (see translator documentation)
- Unsafe for annotation Cases when constructors coercions are used or when annotations are eta-reduced predicates

3.12.4 Details of changes in 8.0

Vernacular commands

- New option ”Set Printing All” to deactivate all high-level forms of printing (implicit arguments, coercions, destructing let, if-then-else, notations, projections)
- ”Functional Scheme” and ”Functional Induction” extended to polymorphic types and dependent types
- Notation now allows recursive patterns, hence recovering parts of the functionalities of pre-V8 Grammar/Syntax commands
- Command ”Print.” discontinued.
- Redundant syntax ”Implicit Arguments On/Off” discontinued

New syntax

- Semantics change of the if-then-else construction in new syntax: ”if c then t1 else t2” now stands for ”match c with c1 _ ... _ => t1 | c2 _ ... _ => t2 end” with no dependency of t1 and t2 in the arguments of the constructors; this may cause incompatibilities for files translated using coq 8.0beta

Interpretation scopes

- Delimiting key %bool for bool_scope added
- Import no more needed to activate argument scopes from a module

Tactics and the tactic Language

- Semantics of ”assert” is now consistent with the reference manual
- New tactics stepl and stepr for chaining transitivity steps
- Tactic ”replace ... with ... in” added
- Intro patterns now supported in Ltac (parsed with prefix ”ipattern:”)

Executables and tools

- Added option -top to change the name of the toplevel module ”Top”
- Coqdoc updated to new syntax and now part of Coq sources
- XML exportation tool now exports the structure of vernacular files (cf chapter 13 in the reference manual)

User contributions

- User contributions have been updated to the new syntax

Bug fixes

- Many bugs have been fixed (cf coq-bugs web page)

CHAPTER

FOUR

THE LANGUAGE

4.1 The Gallina specification language

This chapter describes Gallina, the specification language of Coq. It allows developing 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 *Terms*. The language of commands, called *The Vernacular* is described in Section *The Vernacular*.

In Coq, logical objects are typed to ensure their logical correctness. The rules implemented by the typing algorithm are described in Chapter *Calculus of Inductive Constructions*.

4.1.1 About the grammars in the manual

Grammars are presented in Backus-Naur form (BNF). Terminal symbols are set in black `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`".

4.1.2 Lexical conventions

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

Comments Comments are enclosed between `(*` and `*)`. They can be nested. They can contain any character. However, embedded *`string`* literals must be correctly closed. Comments are treated as blanks.

Identifiers and field 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 grammar (except that the string `_` is reserved; it is not a valid identifier):

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

```
ident              ::=   first_letter[subsequent_letter…subsequent_letter]
field              ::=   .ident
first_letter       ::=   a..z   A..Z   _   unicode_letter
subsequent_letter  ::=   first_letter   0..9   '   unicode_id_part
```

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

Field identifiers, written *field*, are identifiers prefixed by `.` (dot) with no blank between the dot and the identifier. They are used in the syntax of qualified identifiers.

Numerals Numerals are sequences of digits with an optional fractional part and exponent, optionally preceded by a minus sign. *int* is an integer; a numeral without fractional or exponent parts. *num* is a non-negative integer. Underscores embedded in the digits are ignored, for example `1_000_000` is the same as `1000000`.

```
numeral   ::=   num[. num][exp[sign]num]
int       ::=   [-]num
num       ::=   digit…digit
digit     ::=   0..9
exp       ::=   e | E
sign      ::=   + | -
```

Strings Strings begin and end with `"` (double quote). Use `""` to represent a double quote character within a string. In the grammar, strings are identified with `string`.

Keywords The following character sequences are reserved keywords that cannot be used as identifiers:

```
_ Axiom CoFixpoint Definition Fixpoint Hypothesis IF Parameter Prop
SProp Set Theorem Type Variable as at by cofix discriminated else
end exists exists2 fix for forall fun if in lazymatch let match
multimatch return then using where with
```

Note that plugins may define additional keywords when they are loaded.

Other tokens The set of tokens defined at any given time can vary because the *Notation* command can define new tokens. A *Require* command may load more notation definitions, while the end of a *Section* may remove notations. Some notations are defined in the basic library (see *The Coq library*) and are normally loaded automatically at startup time.

Here are the character sequences that Coq directly defines as tokens without using *Notation* (omitting 25 specialized tokens that begin with `#int63_`):

```
! #[ % & ' ( () (bfs) (dfs) ) * ** + , - ->
. .( .. ... / : ::= := :> :>> ; < <+ <- <:
<<: <= = => > >-> >= ? @ @{ [ [= ] _ _eqn
`( `{ { {| | |- || }
```

When multiple tokens match the beginning of a sequence of characters, the longest matching token is used. Occasionally you may need to insert spaces to separate tokens. For example, if `~` and `~~` are both defined as tokens, the inputs `~ ~` and `~~` generate different tokens, whereas if `~~` is not defined, then the two inputs are equivalent.

4.1.3 Terms

Syntax of terms

The following grammars describe the basic syntax of the terms of the *Calculus of Inductive Constructions* (also called Cic). The formal presentation of Cic is given in Chapter *Calculus of Inductive Constructions*. Extensions of this syntax are given in Chapter *Extensions of Gallina*. How to customize the syntax is described in Chapter *Syntax extensions and interpretation scopes*.

```
term          ::=  forall binders , term
                   fun binders => term
                   fix fix_bodies
                   cofix cofix_bodies
                   let ident [binders] [: term] := term in term
                   let fix fix_body in term
                   let cofix cofix_body in term
                   let ( [name , … , name] ) [dep_ret_type] := term in term
                   let ' pattern [in term] := term [return_type] in term
                   if term [dep_ret_type] then term else term
                   term : term
                   term <: term
                   term :>
                   term -> term
                   term arg … arg
                   @ qualid [term … term]
                   term % ident
                   match match_item , … , match_item [return_type] with
                   [[|] equation | … | equation] end
                   qualid
                   sort
                   num
                   _
                   ( term )
arg           ::=  term
                   ( ident := term )
binders       ::=  binder … binder
binder        ::=  name
                   ( name … name : term )
                   ( name [: term] := term )
                   ' pattern
name          ::=  ident | _
qualid        ::=  ident | qualid field
sort          ::=  SProp | Prop | Set | Type
fix_bodies    ::=  fix_body
                   fix_body with fix_body with … with fix_body for ident
cofix_bodies  ::=  cofix_body
                   cofix_body with cofix_body with … with cofix_body for ident
fix_body      ::=  ident binders [annotation] [: term] := term
cofix_body    ::=  ident [binders] [: term] := term
annotation    ::=  { struct ident }
match_item    ::=  term [as name] [in qualid [pattern … pattern]]
dep_ret_type  ::=  [as name] return_type
```

```
return_type   ::=  return term
equation      ::=  mult_pattern | … | mult_pattern => term
mult_pattern  ::=  pattern , … , pattern
pattern       ::=  qualid pattern … pattern
                   @ qualid pattern … pattern
                   pattern as ident
                   pattern % ident
                   qualid
                   _
                   num
                   ( pattern | … | pattern )
```

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*.

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*, while qualified identifiers do not.

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 *Syntax extensions and interpretation scopes* for details). Initially, numerals are bound to Peano's representation of natural numbers (see *Datatypes*).

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

Sorts

There are four sorts `SProp`, `Prop`, `Set` and `Type`.

- `SProp` is the universe of *definitionally irrelevant propositions* (also called *strict propositions*).
- `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 sorts.

More on sorts can be found in Section *Sorts*.

Binders

Various constructions such as `fun`, `forall`, `fix` and `cofix` *bind* variables. A binding is represented by an identifier. If the binding variable is not used in the expression, the identifier can be replaced by the symbol `_`. When the type of a bound variable cannot be synthesized by the system, it can be specified with the notation `(ident : type)`. There is also a notation for a sequence of binding variables sharing the same type: `( ident+ : type)`. A binder can also be any pattern prefixed by a quote, e.g. `'(x,y)`.

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

Lists of `binder`s are allowed. In the case of `fun` and `forall`, it is intended that at least one binder of the list is an assumption otherwise fun and forall gets identical. Moreover, parentheses can be omitted in the case of a single sequence of bindings sharing the same type (e.g.: `fun (x y z : A) => t` can be shortened in `fun x y z : A => t`).

Abstractions

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

Products

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

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

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

Applications

The expression `term`$_{fun}$ `term` denotes the application of `term`$_{fun}$ (which is expected to have a function type) to `term`.

The expression `term`$_{fun}$ `term`$_i$+ denotes the application of the term `term`$_{fun}$ to the arguments `term`$_i$. It is equivalent to `( … (` `term`$_{fun}$ `term`$_1$ `) … )` `term`$_n$: associativity is to the left.

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

Type cast

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

term <: *type* locally sets up the virtual machine for checking that *term* has type *type*.

term <<: *type* uses native compilation for checking that *term* has type *type*.

Inferable subterms

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

Let-in definitions

let *ident* := *term* in *term*' denotes the local binding of *term* to the variable *ident* in *term*'. There is a syntactic sugar for let-in definition of functions: let *ident* *binder*$^+$:= *term* in *term*' stands for let *ident* := fun *binder*$^+$ => *term* in *term*'.

Definition by case analysis

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

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

The expression match "$term_0$ *return_type* with $pattern_1$ => $term_1$ | … | $pattern_n$ => $term_n$ end" denotes a *pattern matching* over the term $term_0$ (expected to be of an inductive type I). The terms $term_1$…$term_n$ are the *branches* of the pattern matching expression. Each of $pattern_i$ has a form *qualid* *ident* where *qualid* must denote a constructor. There should be exactly one branch for every constructor of I.

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

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

```
Inductive bool : Type := true : bool | false : bool.
Inductive eq (A:Type) (x:A) : A -> Prop := eq_refl : eq A x x.
Inductive or (A:Prop) (B:Prop) : Prop :=
  | or_introl : A -> or A B
  | or_intror : B -> or A B.
Definition bool_case (b:bool) : or (eq bool b true) (eq bool b false) :=
  match b as x return or (eq bool x true) (eq bool x false) with
  | true => or_introl (eq bool true true) (eq bool true false) (eq_refl bool true)
```

(continues on next page)

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

(continued from previous page)

```
  | false => or_intror (eq bool false true) (eq bool false false) (eq_refl bool false)
  end.
```

the branches have respective types "`or (eq bool true true) (eq bool true false)`" and "`or (eq bool false true) (eq bool false false)`" while the whole pattern matching expression has type "`or (eq bool b true) (eq bool b false)`", the identifier `b` being used to represent the dependency.

Note: When the term being matched is a variable, the `as` clause can be omitted and the term being matched can serve itself as binding name in the return type. For instance, the following alternative definition is accepted and has the same meaning as the previous one.

```
Definition bool_case (b:bool) : or (eq bool b true) (eq bool b false) :=
match b return or (eq bool b true) (eq bool b false) with
| true => or_introl (eq bool true true) (eq bool true false) (eq_refl bool true)
| false => or_intror (eq bool false true) (eq bool false false) (eq_refl bool false)
end.
```

The second subcase is only relevant for annotated inductive types such as the equality predicate (see Section *Equality*), the order predicate on natural numbers or the type of lists of a given length (see Section *Matching objects of dependent types*). In this configuration, the type of each branch can depend on the type dependencies specific to the branch and the whole pattern matching expression has a type determined by the specific dependencies in the type of the term being matched. This dependency of the return type in the annotations of the inductive type is expressed using a "`in` I `_ … _` *`pattern`*$_1$ … *`pattern`*$_n$" clause, where

- I is the inductive type of the term being matched;
- the `_` are matching the parameters of the inductive type: the return type is not dependent on them.
- the *`pattern`*$_i$ are matching the annotations of the inductive type: the return type is dependent on them
- in the basic case which we describe below, each *`pattern`*$_i$ is a name *`ident`*$_i$; see *Patterns in in* for the general case

For instance, in the following example:

```
Definition eq_sym (A:Type) (x y:A) (H:eq A x y) : eq A y x :=
match H in eq _ _ z return eq A z x with
| eq_refl _ _ => eq_refl A x
end.
```

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

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

There are specific notations for case analysis on types with one or two constructors: `if … then … else …` and `let (…,…) := … in …` (see Sections *Pattern-matching on boolean values: the if expression* and *Irrefutable patterns: the destructuring let variants*).

Recursive functions

The expression "`fix` *ident*$_1$ *binder*$_1$ `:` *type*$_1$ `:=` *term*$_1$ `with … with` *ident*$_n$ *binder*$_n$ `:` *type*$_n$ `:=` *term*$_n$ `for` *ident*$_i$" denotes the i-th component of a block of functions defined by mutual structural recursion. It is the local counterpart of the *Fixpoint* command. When $n = 1$, the "`for` *ident*$_i$" clause is omitted.

The expression "`cofix` *ident*$_1$ *binder*$_1$ `:` *type*$_1$ `with … with` *ident*$_n$ *binder*$_n$ `:` *type*$_n$ `for` *ident*$_i$" denotes the i-th component of a block of terms defined by a mutual guarded co-recursion. It is the local counterpart of the *CoFixpoint* command. When $n = 1$, the "`for` *ident*$_i$" clause is omitted.

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

4.1.4 The Vernacular

```
decorated-sentence   ::=   [ decoration … decoration ] sentence
sentence             ::=   assumption
                           definition
                           inductive
                           fixpoint
                           assertion proof
assumption           ::=   assumption_keyword assums.
assumption_keyword   ::=   Axiom | Conjecture
                           Parameter | Parameters
                           Variable | Variables
                           Hypothesis | Hypotheses
assums               ::=   ident … ident : term
                           ( ident … ident : term ) … ( ident … ident : term )
definition           ::=   [Local] Definition ident [binders] [: term] := term .
                           Let ident [binders] [: term] := term .
inductive            ::=   Inductive ind_body with … with ind_body .
                           CoInductive ind_body with … with ind_body .
ind_body             ::=   ident [binders] : term :=
                           [[|] ident [binders] [:term] | … | ident [binders] [:term]]
fixpoint             ::=   Fixpoint fix_body with … with fix_body .
                           CoFixpoint cofix_body with … with cofix_body .
assertion            ::=   assertion_keyword ident [binders] : term .
assertion_keyword    ::=   Theorem | Lemma
                           Remark | Fact
                           Corollary | Property | Proposition
                           Definition | Example
proof                ::=   Proof . … Qed .
                           Proof . … Defined .
                           Proof . … Admitted .
decoration           ::=   #[ attributes ]
attributes           ::=   [attribute, … , attribute]
attribute            ::=   ident
                           ident = string
                           ident ( attributes )
```

This grammar 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.

Sentences may be *decorated* with so-called *attributes*, which are described in the corresponding section (*At-*

tributes).

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

Assumptions

Assumptions extend the environment with axioms, parameters, hypotheses or variables. An assumption binds an *ident* to a *type*. It is accepted by Coq if and only if this *type* is a correct type in the environment preexisting the declaration and if *ident* was not previously defined in the same module. This *type* is considered to be the type (or specification, or statement) assumed by *ident* and we say that *ident* has type *type*.

Command: `Parameter` *ident* `:` *type*

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

Error: *ident* **already exists.**

Variant: `Parameter` *ident*+ `:` *type*

Adds several parameters with specification *type*.

Variant: `Parameter` (`(` *ident*+ `:` *type* `)`)+

Adds blocks of parameters with different specifications.

Variant: `Local Parameter` (`(` *ident*+ `:` *type* `)`)+

Such parameters are never made accessible through their unqualified name by *Import* and its variants. You have to explicitly give their fully qualified name to refer to them.

Variant: `Local`? `Parameters` (`(` *ident*+ `:` *type* `)`)+

Variant: `Local`? `Axiom` (`(` *ident*+ `:` *type* `)`)+

Variant: `Local`? `Axioms` (`(` *ident*+ `:` *type* `)`)+

Variant: `Local`? `Conjecture` (`(` *ident*+ `:` *type* `)`)+

Variant: `Local`? `Conjectures` (`(` *ident*+ `:` *type* `)`)+

These variants are synonyms of `Local`? `Parameter` (`(` *ident*+ `:` *type* `)`)+ .

Variant: `Variable` (`(` *ident*+ `:` *type* `)`)+

Variant: `Variables` (`(` *ident*+ `:` *type* `)`)+

Variant: `Hypothesis` (`(` *ident*+ `:` *type* `)`)+

Variant: `Hypotheses` (`(` *ident*+ `:` *type* `)`)+

Outside of any section, these variants are synonyms of `Local Parameter`

(*ident*+ : *type*)+ . For their meaning inside a section, see *Variable* in *Section mechanism*.

Warning: `ident is declared as a local axiom [local-declaration,scope]`
Warning generated when using *Variable* instead of *Local Parameter*.

Note: It is advised to use the commands *Axiom*, *Conjecture* and *Hypothesis* (and their plural forms) for logical postulates (i.e. when the assertion *type* is of sort `Prop`), and to use the commands *Parameter* and *Variable* (and their plural forms) in other cases (corresponding to the declaration of an abstract mathematical entity).

See also:

Section *Section mechanism*.

Definitions

Definitions extend the environment with associations of names to terms. A definition can be seen as a way to give a meaning to a name or as a way to abbreviate a term. In any case, the name can later be replaced at any time by its definition.

The operation of unfolding a name into its definition is called δ-conversion (see Section *δ-reduction*). A definition is accepted by the system if and only if the defined term is well-typed in the current context of the definition and if the name is not already used. The name defined by the definition is called a *constant* and the term it refers to is its *body*. A definition has a type which is the type of its body.

A formal presentation of constants and environments is given in Section *Typing rules*.

Command: `Definition ident := term`
This command binds *term* to the name *ident* in the environment, provided that *term* is well-typed.

Error: `ident already exists.`

Variant: `Definition ident : type := term`
This variant checks that the type of *term* is definitionally equal to *type*, and registers *ident* as being of type *type*, and bound to value *term*.

Error: `The term term has type type while it is expected to have type type'.`

Variant: `Definition ident binders [: type]? := term`
This is equivalent to `Definition ident : forall binders, type := fun binders => term`.

Variant: `Local Definition ident [binders]? [: type]? := term`
Such definitions are never made accessible through their unqualified name by *Import* and its variants. You have to explicitly give their fully qualified name to refer to them.

Variant: `[Local]? Example ident [binders]? [: type]? := term`
This is equivalent to *Definition*.

Variant: `Let ident := term`
Outside of any section, this variant is a synonym of `Local Definition ident := term`. For its meaning inside a section, see *Let* in *Section mechanism*.

Warning: `ident is declared as a local definition [local-declaration,scope]`
Warning generated when using *Let* instead of *Local Definition*.

See also:

Section *Section mechanism*, commands *Opaque*, *Transparent*, and tactic *unfold*.

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

Command: `Inductive` *ident* `:` *sort*? `:=` |? *ident* `:` *type* (| *ident* `:` *type*)*

This command defines a simple inductive type and its constructors. The first *ident* is the name of the inductively defined type and *sort* is the universe where it lives. The next *ident*s are the names of its constructors and *type* their respective types. Depending on the universe where the inductive type *ident* lives (e.g. its type *sort*), Coq provides a number of destructors. Destructors are named *ident*`_sind`,:token:`ident_ind`, *ident*`_rec` or *ident*`_rect` which respectively correspond to elimination principles on `SProp`, `Prop`, `Set` and `Type`. The type of the destructors expresses structural induction/recursion principles over objects of type *ident*. 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).

Error: `Non strictly positive occurrence of` *ident* `in` *type*`.`

The types of the constructors have to satisfy a *positivity condition* (see Section *Positivity Condition*). This condition ensures the soundness of the inductive definition. The positivity checking can be disabled using the *Positivity Checking* flag (see *Controlling Typing Flags*).

Error: `The conclusion of` *type* `is not valid; it must be built from` *ident*`.`

The conclusion of the type of the constructors must be the inductive type *ident* being defined (or *ident* applied to arguments in the case of annotated inductive types — cf. next section).

Example

The set of natural numbers is defined as:

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

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

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

```
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 proving 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`.

Variant: `Inductive` *ident* `: `*sort*? `:=` `|`? *ident* *binders*? `: `*type*? * |

Constructors *idents* can come with *binders* in which case, the actual type of the constructor is `forall` *binders*, *type*.

In the case where inductive types have no annotations (next section gives an example of such annotations), a constructor can be defined by only giving the type of its arguments.

Example

```
Inductive nat : Set := O | S (_:nat).
```

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.

Example

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

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

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

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

Parameterized inductive types

Variant: `Inductive` *ident binders* [`:` *type*]? `:=` [`|`]? *ident* `:` *type* [`|` *ident* `:` *type*]*

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

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

Example

A typical example is the definition of polymorphic lists:

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

In the type of `nil` and `cons`, we write `(list A)` and not just `list`. The constructors `nil` and `cons` will have respectively types:

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

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

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

Once again, it is possible to specify only the type of the arguments of the constructors, and to omit the type of the conclusion:

```
Inductive list (A:Set) : Set := nil | cons (_:A) (_:list A).
```

Note:

- It is possible in the type of a constructor, to invoke recursively the inductive definition on an argument which is not the parameter itself.

  One can define :

  ```
  Inductive list2 (A:Set) : Set :=
  | nil2 : list2 A
  | cons2 : A -> list2 (A*A) -> list2 A.
      list2 is defined
      list2_rect is defined
      list2_ind is defined
      list2_rec is defined
      list2_sind is defined
  ```

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

```
Inductive list2 (A:Set) : Set := nil2 | cons2 (_:A) (_:list2 (A*A)).
    list2 is defined
    list2_rect is defined
    list2_ind is defined
    list2_rec is defined
    list2_sind is defined
```

But the following definition will give an error:

```
Fail Inductive listw (A:Set) : Set :=
| nilw : listw (A*A)
| consw : A -> listw (A*A) -> listw (A*A).
    The command has indeed failed with message:
    Last occurrence of "listw" must have "A" as 1st argument in
     "listw (A * A)%type".
```

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

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

Flag: `Uniform Inductive Parameters`

When this flag is set (it is off by default), inductive definitions are abstracted over their parameters before type checking constructors, allowing to write:

```
Set Uniform Inductive Parameters.
Inductive list3 (A:Set) : Set :=
| nil3 : list3
| cons3 : A -> list3 -> list3.
    list3 is defined
    list3_rect is defined
    list3_ind is defined
    list3_rec is defined
    list3_sind is defined
```

This behavior is essentially equivalent to starting a new section and using *`Context`* to give the uniform parameters, like so (cf. *Section mechanism*):

```
Section list3.
Context (A:Set).
    A is declared

Inductive list3 : Set :=
| nil3 : list3
| cons3 : A -> list3 -> list3.
    list3 is defined
    list3_rect is defined
    list3_ind is defined
    list3_rec is defined
    list3_sind is defined

End list3.
```

See also:

Section *Inductive Definitions* and the *`induction`* tactic.

Variants

Command: `Variant` *ident* *binders* `: `*type*? `:=` `|`? *ident* `:` *type* `| `*ident*` : `*type**

The *Variant* command is identical to the *Inductive* command, except that it disallows recursive definition of types (for instance, lists cannot be defined using *Variant*). No induction scheme is generated for this variant, unless the *Nonrecursive Elimination Schemes* flag is on.

Error: `The` *num* `th argument of` *ident* `must be` *ident* `in` *type*`.`

Mutually defined inductive types

Variant: `Inductive` *ident* `: `*type*? `:=` `|`? *ident*` : `*type* `|`* `with` `|`? *ident* `: `*type*? `|`* *

This variant allows defining a block of mutually inductive types. It has the same semantics as the above *Inductive* definition for each *ident*. All *ident* are simultaneously added to the environment. Then well-typing of constructors can be checked. Each one of the *ident* can be used on its own.

Variant: `Inductive` *ident* *binders* `: `*type*? `:=` `|`? *ident*` : `*type* `|`* `with` `|`? *ident* *binders* `: `*type*

In this variant, the inductive definitions are parameterized with *binders*. 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.

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.

```
Parameters A B : Set.
Inductive tree : Set := node : A -> forest -> tree

with forest : Set :=
| leaf : B -> forest
| 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`.

```
Check tree_rec.
    tree_rec
         : forall P : tree -> Set,
           (forall (a : A) (f : forest), P (node a f)) -> forall t : tree, P t

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

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

```
Inductive tree (A B:Set) : Set := node : A -> forest A B -> tree A B

with forest (A B:Set) : Set :=
| leaf : B -> forest A B
| cons : tree A B -> forest A B -> forest A B.
```

Assume we define an inductive definition inside a section (cf. *Section mechanism*). 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:

A generic command `Scheme` is useful to build automatically various mutual induction principles.

Co-inductive types

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

Command: `CoInductive ident binders [: type]? := [|]? ident : type [| ident : type]*`

This command introduces a co-inductive type. The syntax of the command is the same as the command `Inductive`. No principle of induction is derived from the definition of a co-inductive type, since such principles only make sense for inductive types. For co-inductive types, the only elimination principle is case analysis.

Example

An example of a co-inductive type is the type of infinite sequences of natural numbers, usually called streams.

```
CoInductive Stream : Set := Seq : nat -> Stream -> Stream.
```

The usual destructors on streams `hd:Stream->nat` and `tl:Str->Str` can be defined as follows:

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

Definition of co-inductive predicates and blocks of mutually co-inductive definitions are also allowed.

Example

An example of a co-inductive predicate is the extensional equality on streams:

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

In order to prove the extensional equality of two streams `s1` and `s2` we have to construct an infinite proof of equality, that is, an infinite object of type `(EqSt s1 s2)`. We will see how to introduce infinite objects in Section *Definitions of recursive objects in co-inductive types*.

Caveat

The ability to define co-inductive types by constructors, hereafter called *positive co-inductive types*, is known to break subject reduction. The story is a bit long: this is due to dependent pattern-matching which implies propositional η-equality, which itself would require full η-conversion for subject reduction to hold, but full η-conversion is not acceptable as it would make type-checking undecidable.

Since the introduction of primitive records in Coq 8.5, an alternative presentation is available, called *negative co-inductive types*. This consists in defining a co-inductive type as a primitive record type through its projections. Such a technique is akin to the *co-pattern* style that can be found in e.g. Agda, and preserves subject reduction.

The above example can be rewritten in the following way.

```
Set Primitive Projections.
CoInductive Stream : Set := Seq { hd : nat; tl : Stream }.
    Stream is defined
    hd is defined
    tl is defined

CoInductive EqSt (s1 s2: Stream) : Prop := eqst {
  eqst_hd : hd s1 = hd s2;
  eqst_tl : EqSt (tl s1) (tl s2);
}.
    EqSt is defined
    eqst_hd is defined
    eqst_tl is defined
```

Some properties that hold over positive streams are lost when going to the negative presentation, typically when they imply equality over streams. For instance, propositional η-equality is lost when going to the negative presentation. It is nonetheless logically consistent to recover it through an axiom.

```
Axiom Stream_eta : forall s: Stream, s = Seq (hd s) (tl s).
    Stream_eta is declared
```

More generally, as in the case of positive coinductive types, it is consistent to further identify extensional equality of coinductive types with propositional equality:

```
Axiom Stream_ext : forall (s1 s2: Stream), EqSt s1 s2 -> s1 = s2.
    Stream_ext is declared
```

As of Coq 8.9, it is now advised to use negative co-inductive types rather than their positive counterparts.

See also:

Primitive Projections for more information about negative records and primitive projections.

Definition of recursive functions

Definition of functions by recursion over inductive objects

This section describes the primitive form of definition by recursion over inductive objects. See the *Function* command for more advanced constructions.

Command: `Fixpoint` *ident binders* `{struct` *ident*`}`? `:` *type*? `:=` *term*

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

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

The `{struct` *ident*`}` annotation may be left implicit, in this case the system tries successively arguments from left to right until it finds one that satisfies the decreasing condition.

Note:

- Some fixpoints may have several arguments that fit as decreasing arguments, and this choice influences the reduction of the fixpoint. Hence an explicit annotation must be used if the leftmost decreasing argument is not the desired one. Writing explicit annotations can also speed up type checking of large mutual fixpoints.
- In order to keep the strong normalization property, the fixed point reduction will only be performed when the argument in position of the decreasing argument (which type should be in an inductive definition) starts with a constructor.

Example

One can define the addition function as :

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

The match operator matches a value (here `n`) with the various constructors of its (inductive) type. The remaining arguments give the respective values to be returned, as functions of the parameters of the corresponding constructor. Thus here when `n` equals `O` we return `m`, and when `n` equals `(S p)` we return `(S (add p m))`.

The match operator is formally described in Section *The match … with … end construction*. 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:

```
Fail Fixpoint wrongplus (n m:nat) {struct n} : nat :=
match m with
| O => n
| S p => S (wrongplus n p)
end.
    The command has indeed failed with message:
    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".
    Recursive definition is:
    "fun n m : nat => match m with
                      | O => n
                      | S p => S (wrongplus n p)
                      end".
```

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

```
Fixpoint plus (n m:nat) {struct m} : nat :=
match m with
| O => n
| S p => S (plus n p)
end.
    plus is defined
    plus is recursively defined (decreasing on 2nd argument)
```

Example

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.

```
Fixpoint mod2 (n:nat) : nat :=
match n with
| O => O
| S p => match p with
         | O => S O
         | S q => mod2 q
         end
end.
    mod2 is defined
    mod2 is recursively defined (decreasing on 1st argument)
```

Variant: `Fixpoint` *ident* *binders* `{struct` *ident*`}`? `:` *type*? `:=` *term* `with` *ident* *binders* `:` *type*? `:=`

This variant allows defining simultaneously several mutual fixpoints. It is especially useful when defining functions over mutually defined inductive types.

Example

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

```
Fixpoint tree_size (t:tree) : nat :=
match t with
| node a f => S (forest_size f)
end
with forest_size (f:forest) : nat :=
match f with
| leaf b => 1
| cons t f' => (tree_size t + forest_size f')
end.
    tree_size is defined
    forest_size is defined
    tree_size, forest_size are recursively defined
    (decreasing respectively on 1st, 1st arguments)
```

Definitions of recursive objects in co-inductive types

Command: `CoFixpoint` *ident* *binders*? `:` *type*? `:=` *term*

This 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 `O` (see Section *Co-inductive types* for the definition of `Stream`, `hd` and `tl`):

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

```
Fail CoFixpoint filter (p:nat -> bool) (s:Stream) : Stream :=
  if p (hd s) then Seq (hd s) (filter p (tl s)) else filter p (tl s).
    The command has indeed failed with message:
    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)".
    Recursive definition is:
    "fun (p : nat -> bool) (s : Stream) =>
     if p (hd s)
     then {| hd := hd s; tl := filter p (tl s) |}
     else 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:

```
Eval compute in (from 0).
    = (cofix from (n : nat) : Stream := {| hd := n; tl := from (S n) |}) 0
         : Stream

Eval compute in (hd (from 0)).
    = 0
         : nat

Eval compute in (tl (from 0)).
    = (cofix from (n : nat) : Stream := {| hd := n; tl := from (S n) |}) 1
         : Stream
```

Variant: `CoFixpoint` *ident* [*binders*]? [: *type*]? := *term* [with *ident* [*binders*]? : [*type*]? := *term*]*

As in the *Fixpoint* command, it is possible to introduce a block of mutually dependent methods.

Assertions and proofs

An assertion states a proposition (or a type) of which the proof (or an inhabitant of the type) is interactively built using tactics. The interactive proof mode is described in Chapter *Proof handling* and the tactics in Chapter *Tactics*. The basic assertion command is:

Command: `Theorem` *ident* [*binders*]? : *type*

After the statement is asserted, Coq needs a proof. Once a proof of *type* under the assumptions represented by *binders* is given and validated, the proof is generalized into a proof of `forall` *binders*, *type* and the theorem is bound to the name *ident* in the environment.

Error: `The term` *term* `has type` *type* `which should be Set, Prop or Type.`

Error: *ident* `already exists.`

The name you provided is already defined. You have then to choose another name.

Error: `Nested proofs are not allowed unless you turn the Nested Proofs Allowed flag on.`

You are asserting a new statement while already being in proof editing mode. This feature, called nested proofs, is disabled by default. To activate it, turn the *Nested Proofs Allowed* flag on.

Variant: `Lemma` *ident* [*binders*]? : *type*

Variant: `Remark` *ident* [*binders*]? : *type*

Variant: `Fact` *ident* [*binders*]? : *type*

Variant: `Corollary` *ident* [*binders*]? : *type*

Variant: `Proposition` *ident* [*binders*]? : *type*

These commands are all synonyms of `Theorem` *ident* [*binders*]? : type.

Variant: `Theorem` *ident* [*binders*]? : *type* [with *ident* [*binders*]? : *type*]*

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

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

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

Variant: `Definition` *ident* *binders*? `:` *type*

This allows defining a term of type *type* using the proof editing mode. It behaves as *Theorem* but is intended to be used in conjunction with *Defined* in order to define a constant of which the computational behavior is relevant.

The command can be used also with *Example* instead of *Definition*.

See also:

Opaque, *Transparent*, *unfold*.

Variant: `Let` *ident* *binders*? `:` *type*

Like `Definition` *ident* *binders*? `:` *type* except that the definition is turned into a let-in definition generalized over the declarations depending on it after closing the current section.

Variant: `Fixpoint` *ident* *binders* `:` *type* (`with` *ident* *binders* `:` *type*)*

This generalizes the syntax of *Fixpoint* so that one or more bodies can be defined interactively using the proof editing mode (when a body is omitted, its type is mandatory in the syntax). When the block of proofs is completed, it is intended to be ended by *Defined*.

Variant: `CoFixpoint` *ident* *binders*? `:` *type* (`with` *ident* *binders*? `:` *type*)*

This generalizes the syntax of *CoFixpoint* so that one or more bodies can be defined interactively using the proof editing mode.

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 *Tactics*. Besides tactics, there are commands to manage the proof editing mode. They are described in Chapter *Proof handling*.

When the proof is completed it should be validated and put in the environment using the keyword *Qed*.

Note:

1. Several statements can be simultaneously asserted provided the *Nested Proofs Allowed* flag was turned on.
2. Not only other assertions but any vernacular command can be given while in the process of proving a given assertion. In this case, the command is understood as if it would have been given before the statements still to be proved. Nonetheless, this practice is discouraged and may stop working in future versions.
3. Proofs ended by *Qed* are declared opaque. Their content cannot be unfolded (see *Performing computations*), thus realizing some form of *proof-irrelevance*. To be able to unfold a proof, the proof should be ended by *Defined*.
4. *Proof* is recommended but can currently be omitted. On the opposite side, *Qed* (or *Defined*) is mandatory to validate a proof.
5. One can also use *Admitted* in place of *Qed* to turn the current asserted statement into an axiom and exit the proof editing mode.

Attributes

Any vernacular command can be decorated with a list of attributes, enclosed between `#[` (hash and opening square bracket) and `]` (closing square bracket) and separated by commas `,`. Multiple space-separated blocks may be provided.

Each attribute has a name (an identifier) and may have a value. A value is either a *`string`* (in which case it is specified with an equal `=` sign), or a list of attributes, enclosed within brackets.

Some attributes are specific to a command, and so are described with that command. Currently, the following attributes are recognized by a variety of commands:

`universes(monomorphic)`, `universes(polymorphic)` Equivalent to the `Monomorphic` and `Polymorphic` flags (see *Polymorphic Universes*).

`program` Takes no value, equivalent to the `Program` flag (see *Program*).

`global`, `local` Take no value, equivalent to the `Global` and `Local` flags (see *Controlling the locality of commands*).

`deprecated` Takes as value the optional attributes `since` and `note`; both have a string value.

This attribute is supported by the following commands: *`Ltac`*, *`Tactic Notation`*, *`Notation`*, *`Infix`*.

It can trigger the following warnings:

Warning: `Tactic` *`qualid`* `is deprecated since` *`string`*`.` *`string`*`.`

Warning: `Tactic Notation` *`qualid`* `is deprecated since` *`string`*`.` *`string`*`.`

Warning: `Notation` *`string`*$_1$ `is deprecated since` *`string`*$_2$`.` *`string`*$_3$`.`

`string`$_1$ is the actual notation, *`string`*$_2$ is the version number, *`string`*$_3$ is the note.

Example

```
From Coq Require Program.
    [Loading ML file extraction_plugin.cmxs ... done]

#[program] Definition one : nat := S _.
    one has type-checked, generating 1 obligation
    Solving obligations automatically...
    1 obligation remaining
    Obligation 1 of one: nat.

Next Obligation.
    1 subgoal

      ============================
      nat

exact O.
    No more subgoals.

Defined.
#[deprecated(since="8.9.0", note="Use idtac instead.")]
Ltac foo := idtac.
    foo is defined

Goal True.
    1 subgoal
```

(continues on next page)

(continued from previous page)

```
      ============================
      True

Proof.
now foo.
    Toplevel input, characters 4-7:
    > now foo.
    >     ^^^
    Warning: Tactic foo is deprecated since 8.9.0. Use idtac instead.
    [deprecated-tactic,deprecated]
    No more subgoals.

Abort.
```

Warning: `Unsupported attribute`

This warning is an error by default. It is caused by using a command with some attribute it does not understand.

4.2 Extensions of Gallina

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

4.2.1 Record types

The *Record* construction is a macro allowing the definition of records as is done in many programming languages. Its syntax is described in the grammar below. 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 defining "signatures".

```
record          ::=  record_keyword record_body with … with record_body
record_keyword  ::=  Record | Inductive | CoInductive
record_body     ::=  ident [ binders ] [: sort ] := [ ident ] { [ field ; … ; field ] }.
field           ::=  ident [ binders ] : type [ where notation ]
                     ident [ binders ] [: type ] := term
```

Command: `Record` *ident binders* [`:` *sort*]? `:=` [*ident*]? `{` [*ident binders* `:` *type*]*; `}`

The first identifier *ident* is the name of the defined record and *sort* is its type. The optional identifier following `:=` is the name of its constructor. If it is omitted, the default name `Build_`*ident*, where *ident* is the record name, is used. If *sort* is omitted, the default sort is Type. The identifiers inside the brackets are the names of fields. For a given field *ident*, its type is `forall` *binders*`,` *type*. Remark that the type of a particular identifier may depend on a previously-given identifier. Thus the order of the fields is important. Finally, *binders* are parameters of the record.

More generally, a record may have explicitly defined (a.k.a. manifest) fields. For instance, we might have: `Record` *ident binders* `:` *sort* `:= {` $ident_1$ `:` $type_1$ `;` $ident_2$ `:=` $term_2$ `;` $ident_3$ `:` $type_3$ `}.` in which case the correctness of $type_3$ may rely on the instance $term_2$ of $ident_2$ and $term_2$ may in turn depend on $ident_1$.

Example

The set of rational numbers may be defined as:

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

Note here that the fields `Rat_bottom_cond` depends on the field `bottom` and `Rat_irred_cond` depends on both `top` and `bottom`.

Let us now see the work done by the `Record` macro. First the macro generates a variant type definition with just one constructor: `Variant` *ident* *binders*? : *sort* := *ident*$_0$ *binders*? .

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

Example

Let us define the rational 1/2:

```
Theorem one_two_irred : forall x y z:nat, x * y = 1 /\ x * z = 2 -> x = 1.
Admitted.
Definition half := mkRat true 1 2 (O_S 1) one_two_irred.
Check half.
```

```
record_term  ::=  {| [field_def ; … ; field_def] |}
field_def    ::=  ident [binders] := term
```

Alternatively, the following syntax allows creating objects by using named fields, as shown in this grammar. The fields do not have to be in any particular order, nor do they have to be all present if the missing ones can be inferred or prompted for (see *Program*).

```
Definition half' :=
  {| sign := true;
     Rat_bottom_cond := O_S 1;
     Rat_irred_cond := one_two_irred |}.
    half' is defined
```

The following settings let you control the display format for types:

Flag: `Printing Records`
If set, use the record syntax (shown above) as the default display format.

You can override the display format for specified types by adding entries to these tables:

Table: `Printing Record` *qualid*

Specifies a set of qualids which are displayed as records. Use the *Add @table* and *Remove @table* commands to update the set of qualids.

Table: `Printing Constructor` *qualid*

Specifies a set of qualids which are displayed as constructors. Use the *Add @table* and *Remove @table* commands to update the set of qualids.

This syntax can also be used for pattern matching.

```
Eval compute in (
  match half with
  | {| sign := true; top := n |} => n
  | _ => 0
  end).
    = 1
         : nat
```

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

```
Eval compute in top half.
    = 1
         : nat

Eval compute in bottom half.
    = 2
         : nat

Eval compute in Rat_bottom_cond half.
    = O_S 1
         : 0 <> bottom half
```

An alternative syntax for projections based on a dot notation is available:

```
Eval compute in half.(top).
    = 1
         : nat
```

Flag: `Printing Projections`

This flag activates the dot notation for printing.

Example

```
Set Printing Projections.
Check top half.
    half.(top)
         : nat
```

```
projection  ::=   term `.` ( qualid )
                  term `.` ( qualid arg … arg )
                  term `.` ( @qualid term … term )
```

Syntax of Record projections

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

Note: Records defined with the `Record` keyword are not allowed to be recursive (references to the record's name in the type of its field raises an error). To define recursive records, one can use the `Inductive` and `CoInductive` keywords, resulting in an inductive or co-inductive record. Definition of mutually inductive or co-inductive records are also allowed, as long as all of the types in the block are records.

Note: Induction schemes are automatically generated for inductive records. Automatic generation of induction schemes for non-recursive records defined with the `Record` keyword can be activated with the *Nonrecursive Elimination Schemes* flag (see *Generation of induction principles with Scheme*).

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

Warning: *ident* **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:

1. The name *ident* already exists in the environment (see *Axiom*).
2. The body of *ident* uses an incorrect elimination for *ident* (see *Fixpoint* and *Destructors*).
3. The type of the projections *ident* depends on previous projections which themselves could not be defined.

Error: `Records declared with the keyword Record or Structure cannot be recursive.`

The record name *ident* appears in the type of its fields, but uses the keyword `Record`. Use the keyword `Inductive` or `CoInductive` instead.

Error: `Cannot handle mutually (co)inductive records.`

Records cannot be defined as part of mutually inductive (or co-inductive) definitions, whether with records only or mixed with standard definitions.

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

See also:

Coercions and records in section *Classes as Records* of the chapter devoted to coercions.

Primitive Projections

Flag: `Primitive Projections`

Turns on the use of primitive projections when defining subsequent records (even through the `Inductive` and `CoInductive` commands). Primitive projections extended the Calculus of Inductive Constructions with a new binary term constructor `r.(p)` representing a primitive projection `p` applied to a record object `r` (i.e., primitive projections are always applied). Even if the record type has parameters, these do not appear in the internal representation of applications of the projection, considerably reducing the sizes of terms when manipulating parameterized records and type checking time. On the user level, primitive projections can be used as a replacement for the usual defined ones, although there are a few notable differences.

Flag: `Printing Primitive Projection Parameters`

This compatibility flag reconstructs internally omitted parameters at printing time (even though they are absent in the actual AST manipulated by the kernel).

Primitive Record Types

When the *Primitive Projections* flag is on, definitions of record types change meaning. When a type is declared with primitive projections, its `match` construct is disabled (see *Primitive Projections* though). To eliminate the (co-)inductive type, one must use its defined primitive projections.

For compatibility, the parameters still appear to the user when printing terms even though they are absent in the actual AST manipulated by the kernel. This can be changed by unsetting the *Printing Primitive Projection Parameters* flag.

There are currently two ways to introduce primitive records types:

1. Through the `Record` command, in which case the type has to be non-recursive. The defined type enjoys eta-conversion definitionally, that is the generalized form of surjective pairing for records: `r = Build_R (r.(`p_1`) … r.(`p_n`))`. Eta-conversion allows to define dependent elimination for these types as well.
2. Through the `Inductive` and `CoInductive` commands, when the body of the definition is a record declaration of the form `Build_R {` p_1 `:` t_1`; … ;` p_n `:` t_n `}`. In this case the types can be recursive and eta-conversion is disallowed. These kind of record types differ from their traditional versions in the sense that dependent elimination is not available for them and only non-dependent case analysis can be defined.

Reduction

The basic reduction rule of a primitive projection is p_i `(Build_R` t_1 … t_n`)` $\rightarrow_\iota$ t_i. However, to take the δ flag into account, projections can be in two states: folded or unfolded. An unfolded primitive projection application obeys the rule above, while the folded version delta-reduces to the unfolded version. This allows to precisely mimic the usual unfolding rules of constants. Projections obey the usual `simpl` flags of the `Arguments` command in particular. There is currently no way to input unfolded primitive projections at the user-level, and there is no way to display unfolded projections differently from folded ones.

Compatibility Projections and `match`

To ease compatibility with ordinary record types, each primitive projection is also defined as a ordinary constant taking parameters and an object of the record type as arguments, and whose body is an application of the unfolded primitive projection of the same name. These constants are used when elaborating partial applications of the projection. One can distinguish them from applications of the primitive projection if the *Printing Primitive Projection Parameters* flag is off: For a primitive projection application, parameters are printed as underscores while for the compatibility projections they are printed as usual.

Additionally, user-written `match` constructs on primitive records are desugared into substitution of the projections, they cannot be printed back as `match` constructs.

4.2.2 Variants and extensions of `match`

Multiple and nested pattern matching

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

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

See also:

Extended pattern matching.

Pattern-matching on boolean values: the if expression

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

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

can be alternatively written

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

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

```
if term [dep_ret_type] then term₁ else term₂
match term [dep_ret_type] with
| C₁ _ … _ => term₁
| C₂ _ … _ => term₂
end
```

Example

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

Notice that the printing uses the `if` syntax because `sumbool` is declared as such (see *Controlling pretty-printing of match expressions*).

Irrefutable patterns: the destructuring let variants

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

First destructuring let syntax

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

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

can be alternatively written

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

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

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

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

```
let (ident_1, …, ident ) [dep_ret_type] := term in term'
```

and

```
match term [dep_ret_type] with
C ident_1 … ident  => term'
end
```

Second destructuring let syntax

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

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

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

```
Definition deep_tuple (A:Set) (x:(A*A)*(A*A)) : A*A*A*A :=
let '((a,b), (c, d)) := x in (a,b,c,d).
    deep_tuple is defined
```

```
Notation " x 'With' p " := (exist _ x p) (at level 20).
    Identifier 'With' now a keyword
```

(continues on next page)

(continued from previous page)

```
Definition proj1_sig' (A:Set) (P:A->Prop) (t:{ x:A | P x }) : A :=
let 'x With p := t in x.
    proj1_sig' is defined
```

When printing definitions which are written using this construct it takes precedence over let printing directives for the datatype under consideration (see Section *Controlling pretty-printing of match expressions*).

Controlling pretty-printing of match expressions

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

Printing nested patterns

Flag: Printing Matching

The Calculus of Inductive Constructions knows pattern matching only over simple patterns. It is however convenient to re-factorize nested pattern matching into a single pattern matching over a nested pattern.

When this flag is on (default), Coq's printer tries to do such limited re-factorization. Turning it off tells Coq to print only simple pattern matching problems in the same way as the Coq kernel handles them.

Factorization of clauses with same right-hand side

Flag: Printing Factorizable Match Patterns

When several patterns share the same right-hand side, it is additionally possible to share the clauses using disjunctive patterns. Assuming that the printing matching mode is on, this flag (on by default) tells Coq's printer to try to do this kind of factorization.

Use of a default clause

Flag: Printing Allow Match Default Clause

When several patterns share the same right-hand side which do not depend on the arguments of the patterns, yet an extra factorization is possible: the disjunction of patterns can be replaced with a `_` default clause. Assuming that the printing matching mode and the factorization mode are on, this flag (on by default) tells Coq's printer to use a default clause when relevant.

Printing of wildcard patterns

Flag: Printing Wildcard

Some variables in a pattern may not occur in the right-hand side of the pattern matching clause. When this flag is on (default), the variables having no occurrences in the right-hand side of the pattern matching clause are just printed using the wildcard symbol "_".

Printing of the elimination predicate

Flag: Printing Synth

In most of the cases, the type of the result of a matched term is mechanically synthesizable. Especially, if the result type does not depend of the matched term. When this flag is on (default), the result type is not printed when Coq knows that it can re- synthesize it.

Printing matching on irrefutable patterns

If an inductive type has just one constructor, pattern matching can be written using the first destructuring let syntax.

Table: `Printing Let` *qualid*

Specifies a set of qualids for which pattern matching is displayed using a let expression. Note that this only applies to pattern matching instances entered with `match`. It doesn't affect pattern matching explicitly entered with a destructuring `let`. Use the *Add @table* and *Remove @table* commands to update this set.

Printing matching on booleans

If an inductive type is isomorphic to the boolean type, pattern matching can be written using `if` … `then` … `else` …. This table controls which types are written this way:

Table: `Printing If` *qualid*

Specifies a set of qualids for which pattern matching is displayed using `if` … `then` … `else` …. Use the *Add @table* and *Remove @table* commands to update this set.

This example emphasizes what the printing settings offer.

Example

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

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

Print snd.
    snd =
    fun (A B : Set) (H : A * B) => let (_, y) := H in y
         : forall A B : Set, A * B -> B

    Arguments snd (_ _)%type_scope

Remove Printing Let prod.
Unset Printing Synth.
Unset Printing Wildcard.
Print snd.
    snd =
    fun (A B : Set) (H : A * B) => match H return B with
                                   | (x, y) => y
                                   end
         : forall A B : Set, A * B -> B

    Arguments snd (_ _)%type_scope
```

4.2.3 Advanced recursive functions

The following experimental command is available when the `FunInd` library has been loaded via `Require Import FunInd`:

Command: `Function` *ident* *binder** `{` *decrease_annot* `}` `:` *type* `:=` *term*

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

```
decrease_annot  ::=  struct ident
                     measure term ident
                     wf term ident
```

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

See the documentation of functional induction (*function induction*) and `Functional Scheme` (*Generation of induction principles with Functional Scheme*) for how to use the induction principle to easily reason about the function.

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

```
Function plus (m n : nat) {struct n} : nat :=
match n with
| 0 => m
| S p => S (plus m p)
end.
    plus is defined
    plus is recursively defined (decreasing on 2nd argument)
    plus_equation is defined
    plus_rect is defined
    plus_ind is defined
    plus_rec is defined
    R_plus_correct is defined
    R_plus_complete is defined
```

than like this:

```
Function plus (n m : nat) {struct n} : nat :=
match n with
| 0 => m
| S p => S (plus p m)
end.
    plus is defined
    plus is recursively defined (decreasing on 1st argument)
    plus_equation is defined
    plus_rect is defined
    plus_ind is defined
```

(continues on next page)

(continued from previous page)

```
plus_rec is defined
R_plus_correct is defined
R_plus_complete is defined
```

Limitations

term must be built as a *pure pattern matching tree* (`match … with`) with applications only *at the end* of each branch.

Function does not support partial application of the function being defined. Thus, the following example cannot be accepted due to the presence of partial application of `wrong` in the body of `wrong`:

```
Function wrong (C:nat) : nat :=
  List.hd 0 (List.map wrong (C::nil)).
    Toplevel input, characters 0-70:
    > Function wrong (C:nat) : nat :=   List.hd 0 (List.map wrong (C::nil)).
    > ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    Error:
    Recursive definition of wrong is ill-formed.
    In environment
    wrong : nat -> nat
    C : nat
    Recursive call to wrong has principal argument equal to
    "C" instead of a subterm of "C".
    Recursive definition is: "fun C : nat => List.hd 0 (List.map wrong [C])".
```

For now, dependent cases are not treated for non structurally terminating functions.

Error: `The recursive argument must be specified.`

Error: `No argument name` *ident*`.`

Error: `Cannot use mutual definition with well-founded recursion or measure.`

Warning: `Cannot define graph for` *ident*`.`

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

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

Warning: `Cannot define principle(s) for` *ident*`.`

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

Warning: `Cannot build functional inversion principle.`

functional inversion will not be available for the function.

See also:

Generation of induction principles with Functional Scheme and *function induction*

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

Variant: `Function` *ident* *binder** `:` *type* `:=` *term*

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

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

Variant: Function *ident* *binder** { struct *ident* } : *type* := *term*

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

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

Variant: Function *ident* *binder** { measure *term* *ident* } : *type* := *term*

Variant: Function *ident* *binder** { wf *term* *ident* } : *type* := *term*

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

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

If the annotation is measure or fw, the user is left with some proof obligations that will be used to define the function. These proofs are: proofs that each recursive call is actually decreasing with respect to the given criteria, and (if the criteria is wf) a proof that the ordering relation is well-founded. Once proof obligations are discharged, the following objects are defined:

- The same objects as with the struct;
- The lemma $\mathtt{ident}_{tcc}$ which collects all proof obligations in one property;
- The lemmas $\mathtt{ident}_{terminate}$ and $\mathtt{ident}_{F}$ which is needed to be inlined during extraction of ident.

The way this recursive function is defined is the subject of several papers by Yves Bertot and Antonia Balaa on the one hand, and Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu on the other hand. Remark: Proof obligations are presented as several subgoals belonging to a Lemma $\mathtt{ident}_{tcc}$.

4.2.4 Section mechanism

Sections create local contexts which can be shared across multiple definitions.

Example

Sections are opened by the *Section* command, and closed by *End*.

```
Section s1.
```

Inside a section, local parameters can be introduced using *Variable*, *Hypothesis*, or *Context* (there are also plural variants for the first two).

```
Variables x y : nat.
    x is declared
    y is declared
```

The command *Let* introduces section-wide *Let-in definitions*. These definitions won't persist when the section is closed, and all persistent definitions which depend on `y'` will be prefixed with `let y' := y in`.

```
Let y' := y.
Definition x' := S x.
Definition x'' := x' + y'.


Print x'.
    x' = S x
         : nat

Print x''.
    x'' = x' + y'
         : nat

End s1.
Print x'.
    x' = fun x : nat => S x
         : nat -> nat

    Arguments x' _%nat_scope

Print x''.
    x'' = fun x y : nat => let y' := y in x' x + y'
         : nat -> nat -> nat

    Arguments x'' (_ _)%nat_scope
```

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

Command: `Section` *ident*

This command is used to open a section named *ident*. Section names do not need to be unique.

Command: `End` *ident*

This command closes the section named *ident*. After closing of the section, the local declarations (variables and local definitions, see *Variable*) get *discharged*, meaning that they stop being visible and that all global objects defined in the section are generalized with respect to the variables and local definitions they each depended on in the section.

Error: `This is not the last opened section.`

Note: Most commands, like *Hint*, *Notation*, option management, … which appear inside a section are canceled when the section is closed.

Command: `Variable` *ident* `:` *type*

This command links *type* to the name *ident* in the context of the current section. 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*).

Error: *ident* `already exists.`

Variant: `Variable` *ident*+ `:` *type*

Links *type* to each *ident*.

Variant: `Variable` (`(` *ident*+ `:` *type* `)`)+

Declare one or more variables with various types.

Variant: `Variables` (`(` *ident*+ `:` *type*`)`)+

Variant: `Hypothesis` (`(` *ident*+ `:` *type*`)`)+

Variant: `Hypotheses` (`(` *ident*+ `:` *type*`)`)+

These variants are synonyms of `Variable` (`(` *ident*+ `:` *type*`)`)+ .

Command: `Let` *ident* `:=` *term*

This command binds the value *term* to the name *ident* in the environment of the current section. The name *ident* is accessible only within the current section. When the section is closed, all persistent definitions and theorems within it and depending on *ident* will be prefixed by the let-in definition `let` *ident* `:=` *term* `in`.

Error: *ident* `already exists.`

Variant: `Let` *ident* (*binders*)? (`:` *type*)? `:=` *term*

Variant: `Let Fixpoint` *ident* *fix_body* (`with` *fix_body*)*

Variant: `Let CoFixpoint` *ident* *cofix_body* (`with` *cofix_body*)*

Command: `Context` *binders*

Declare variables in the context of the current section, like *Variable*, but also allowing implicit variables, *Implicit generalization*, and let-binders.

```
Context {A : Type} (a b : A).
Context `{EqDec A}.
Context (b' := b).
```

See also:

Section *Binders*. Section *Sections and contexts* in chapter *Typeclasses*.

4.2.5 Module system

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

```
module_type        ::=   qualid
                         module_type with Definition qualid := term
                         module_type with Module qualid := qualid
                         qualid qualid … qualid
                         !qualid qualid … qualid
module_binding     ::=   ( [Import|Export] ident … ident : module_type )
```

```
module_bindings    ::=   module_binding … module_binding
module_expression  ::=   qualid … qualid
                         !qualid … qualid
```

Syntax of modules

In the syntax of module application, the ! prefix indicates that any `Inline` directive in the type of the functor arguments will be ignored (see the *`Module Type`* command below).

Command: `Module` *ident*

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

Variant: `Module` *ident* *module_binding**

Starts an interactive functor with parameters given by module_bindings.

Variant: `Module` *ident* `:` *module_type*

Starts an interactive module specifying its module type.

Variant: `Module` *ident* *module_binding** `:` *module_type*

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

Variant: `Module` *ident* `<:` *module_type*+`<:`

Starts an interactive module satisfying each *module_type*.

Variant: `Module` *ident* *module_binding** `<:` *module_type*+`<:` .

Starts an interactive functor with parameters given by the list of *module_binding*. The output module type is verified against each *module_type*.

Variant: `Module` `Import` | `Export`

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

Reserved commands inside an interactive module

Command: `Include` *module*

Includes the content of module in the current interactive module. Here module can be a module expression or a module type expression. If module is a high-order module or module type expression then the system tries to instantiate module by the current interactive module.

Command: `Include` *module*+`<+`

is a shortcut for the commands `Include` *module* for each `module`.

Command: `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: `No such label` *ident*`.`

Error: `Signature components for label` *ident* `do not match.`

Error: `This is not the last opened module.`

Command: `Module` *ident* `:=` *module_expression*

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

Variant: `Module ident module_binding* := module_expression`

Defines a functor with parameters given by the list of *module_binding* and body *module_expression*.

Variant: `Module ident module_binding* : module_type := module_expression`

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

Variant: `Module ident module_binding* <: module_type+<: := module_expression`

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

Variant: `Module ident module_binding* := module_expression+<+`

is equivalent to an interactive module where each *module_expression* is included.

Command: `Module Type ident`

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

Variant: `Module Type ident module_binding*`

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

Reserved commands inside an interactive module type:

Command: `Include module`

Same as `Include` inside a module.

Command: `Include module+<+`

This is a shortcut for the command `Include` *module* for each `module`.

Command: `assumption_keyword Inline assums`

The instance of this assumption will be automatically expanded at functor application, except when this functor application is prefixed by a `!` annotation.

Command: `End ident`

This command closes the interactive module type *ident*.

Error: `This is not the last opened module type.`

Command: `Module Type ident := module_type`

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

Variant: `Module Type ident module_binding* := module_type`

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

Variant: `Module Type ident module_binding* := module_type+<+`

is equivalent to an interactive module type were each *module_type* is included.

Command: `Declare Module ident : module_type`

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

Variant: `Declare Module ident module_binding* : module_type`

Declares a functor with parameters given by the list of *module_binding* and output module type *module_type*.

Example

Let us define a simple module.

```
Module M.
    Interactive Module M started

Definition T := nat.
    T is defined

Definition x := 0.
    x is defined

Definition y : bool.
    1 subgoal

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

exact true.
    No more subgoals.

Defined.
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:

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

A simple module type:

```
Module Type SIG.
    Interactive Module Type SIG started

Parameter T : Set.
    T is declared

Parameter x : T.
    x is declared

End SIG.
    Module Type SIG is defined
```

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.

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

Print N.T.
    N.T = nat
         : Set
```

(continues on next page)

(continued from previous page)

```
Print N.x.
    *** [ N.x : N.T ]

Fail Print N.y.
    The command has indeed failed with message:
    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:

```
Module Type SIG'.
    Interactive Module Type SIG' started

Definition T : Set := nat.
    T is defined

Parameter x : T.
    x is declared

End SIG'.
    Module Type SIG' is defined

Module N : SIG' := M.
    Module N is defined
```

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

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

Print P.y.
    P.y = true
         : bool
```

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

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

Definition T := (X.T * Y.T)%type.
    T is defined

Definition x := (X.x, Y.x).
    x is defined

End Two.
    Module Two is defined
```

and apply it to our modules and do some computations:

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

Eval compute in (fst Q.x + snd Q.x).
```

(continues on next page)

(continued from previous page)

```
    = 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:

```
Module Type SIG2.
    Interactive Module Type SIG2 started

Declare Module M1 : SIG.
    Module M1 is declared

Module M2 <: SIG.
    Interactive Module M2 started

Definition T := M1.T.
    T is defined

Parameter x : T.
    x is declared

End M2.
    Module M2 is defined

End SIG2.
    Module Type SIG2 is defined

Module Mod <: SIG2.
    Interactive Module Mod started

Module M1.
    Interactive Module M1 started

Definition T := nat.
    T is defined

Definition x := 1.
    x is defined

End M1.
    Module M1 is defined

Module M2 := M.
    Module M2 is defined

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.

Note:

1. Modules and module types can be nested components of each other.
2. One can have sections inside a module or a module type, but not a module or a module type inside a section.

3. 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`.

Command: `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

```
Module Mod.
    Interactive Module Mod started

Definition T:=nat.
    T is defined

Check T.
    T
         : Set

End Mod.
    Module Mod is defined

Check Mod.T.
    Mod.T
         : Set

Fail Check T.
    The command has indeed failed with message:
    The reference T was not found in the current environment.

Import Mod.
Check T.
    T
         : Set
```

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

Declarations made with the `Local` flag are never imported by the *Import* command. Such declarations are only accessible through their fully qualified name.

Example

```
Module A.
    Interactive Module A started

Module B.
```

(continues on next page)

(continued from previous page)

```
    Interactive Module B started

Local Definition T := nat.
    T is defined

End B.
    Module B is defined

End A.
    Module A is defined

Import A.
Fail Check B.T.
    The command has indeed failed with message:
    The reference B.T was not found in the current environment.
```

Variant: `Export` *`qualid`*

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

Error: *`qualid`* **`is not a module.`**

Warning: **`Trying to mask the absolute name`** *`qualid`***`!`**

Command: `Print Module` *`ident`*

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

Command: `Print Module Type` *`ident`*

Prints the module type corresponding to *`ident`*.

Flag: `Short Module Printing`

This flag (off by default) disables the printing of the types of fields, leaving only their names, for the commands *`Print Module`* and *`Print Module Type`*.

4.2.6 Libraries and qualified names

Names of libraries

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

Qualified names

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

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

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

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

Visibility

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

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

Example

```
Check 0.
    0
         : nat

Definition nat := bool.
    nat is defined

Check 0.
    0
         : Datatypes.nat

Check Datatypes.nat.
    Datatypes.nat
         : Set

Locate nat.
    Constant Top.nat
    Inductive Coq.Init.Datatypes.nat
      (shorter name to refer to it in current context is Datatypes.nat)
```

See also:

Commands *`Locate`* and *`Locate Library`*.

Libraries and filesystem

Note: The questions described here have been subject to redesign in Coq 8.5. Former versions of Coq use the same terminology to describe slightly different things.

Compiled files (`.vo` and `.vio`) store sub-libraries. In order to refer to them inside Coq, a translation from file-system names to Coq names is needed. In this translation, names in the file system are called *physical* paths while Coq names are contrastingly called *logical* names.

A logical prefix Lib can be associated with a physical path using the command line option `-Q path Lib`. All subfolders of path are recursively associated to the logical path `Lib` extended with the corresponding suffix coming from the physical path. For instance, the folder `path/fOO/Bar` maps to `Lib.fOO.Bar`. Subdirectories corresponding to invalid Coq identifiers are skipped, and, by convention, subdirectories named `CVS` or `_darcs` are skipped too.

Thanks to this mechanism, `.vo` files are made available through the logical name of the folder they are in, extended with their own basename. For example, the name associated to the file `path/fOO/Bar/File.vo` is `Lib.fOO.Bar.File`. The same caveat applies for invalid identifiers. When compiling a source file, the `.vo` file stores its logical name, so that an error is issued if it is loaded with the wrong loadpath afterwards.

Some folders have a special status and are automatically put in the path. Coq commands associate automatically a logical path to files in the repository trees rooted at the directory from where the command is launched, `coqlib/user-contrib/`, the directories listed in the `$COQPATH`, `${XDG_DATA_HOME}/coq/` and `${XDG_DATA_DIRS}/coq/` environment variables (see XDG base directory specification[294]) with the same physical-to-logical translation and with an empty logical prefix.

The command line option `-R` is a variant of `-Q` which has the strictly same behavior regarding loadpaths, but which also makes the corresponding `.vo` files available through their short names in a way not unlike the `Import` command (see *here*). For instance, `-R path Lib` associates to the file `/path/fOO/Bar/File.vo` the logical name `Lib.fOO.Bar.File`, but allows this file to be accessed through the short names `fOO.Bar.File,Bar.File` and `File`. If several files with identical base name are present in different subdirectories of a recursive loadpath, which of these files is found first may be system- dependent and explicit qualification is recommended. The `From` argument of the `Require` command can be used to bypass the implicit shortening by providing an absolute root to the required file (see *Compiled files*).

There also exists another independent loadpath mechanism attached to OCaml object files (`.cmo` or `.cmxs`) rather than Coq object files as described above. The OCaml loadpath is managed using the option `-I path` (in the OCaml world, there is neither a notion of logical name prefix nor a way to access files in subdirectories of path). See the command *`Declare ML Module`* in *Compiled files* to understand the need of the OCaml loadpath.

See *By command line options* for a more general view over the Coq command line options.

4.2.7 Implicit arguments

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

The different kinds of implicit arguments

[294] http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html

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

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

Strict Implicit Arguments

An implicit argument can be either strict or non strict. An implicit argument is said to be *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`. Also, the first argument of `cons` is strict with respect to the second one, since the first argument is exactly the type of the second argument. On the contrary, the second argument of a term of type

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

is implicit but not strict, since it can only be inferred from the type `P n` of the third argument and if `P` is, e.g., `fun _ => True`, it reduces to an expression where `n` does not occur any longer. The first argument `P` is implicit but not strict either because it can only be inferred from `P n` and `P` is not canonically inferable from an arbitrary `n` and the normal form of `P n`. Consider, e.g., that `n` is 0 and the third argument has type `True`, then any `P` of the form

```
fun n => match n with 0 => True | _ => anything end
```

would be a solution of the inference problem.

Contextual Implicit Arguments

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

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

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

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

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

Reversible-Pattern Implicit Arguments

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

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

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

See *Controlling reversible-pattern implicit arguments* for the automatic declaration of reversible-pattern implicit arguments.

Implicit arguments inferable by resolution

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

Maximal or non maximal insertion of implicit arguments

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

Each implicit argument can be declared to have to be inserted maximally or non maximally. This can be governed argument per argument by the command *`Arguments (implicits)`* or globally by the *`Maximal Implicit Insertion`* flag.

See also:

Displaying what the implicit arguments are.

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: **`Cannot infer a term for this placeholder.`**

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

Declaration of implicit arguments

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

Implicit Argument Binders

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

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

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

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

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

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

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

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

Print list.
    Inductive list (A : Type) : Type :=  nil : list | cons : A -> list -> list

    Arguments list {A}%type_scope
    Arguments nil {A}%type_scope
    Arguments cons {A}%type_scope
```

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

Declaring Implicit Arguments

Command: `Arguments` *qualid* `[` *ident* `]` `{` *ident* `}` *ident* `*`

This command is used to set implicit arguments *a posteriori*, where the list of possibly bracketed *ident* is a prefix of the list of arguments of *qualid* where the ones to be declared implicit are surrounded by square brackets and the ones to be declared as maximally inserted implicits are surrounded by curly braces.

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

Command: `Arguments` *qualid* `: clear implicits`

This command clears implicit arguments.

Variant: `Global Arguments` *qualid* `[` *ident* `]` `{` *ident* `}` *ident* `*`

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

Variant: `Local Arguments` *qualid* ([*ident*] | { *ident* } | *ident*)*

When in a module, tell not to activate the implicit arguments of *qualid* declared by this command to contexts that require the module.

Variant: (`Global` | `Local`)? `Arguments` *qualid* (([*ident*] | { *ident* } | *ident*)+),*

For names of constants, inductive types, constructors, lemmas which can only be applied to a fixed number of arguments (this excludes for instance constants whose type is polymorphic), multiple implicit arguments declarations can be given. Depending on the number of arguments qualid is applied to in practice, the longest applicable list of implicit arguments is used to select which implicit arguments are inserted. For printing, the omitted arguments are the ones of the longest list of implicit arguments of the sequence.

Example

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A -> list A -> list A.
    list is defined
    list_rect is defined
    list_ind is defined
    list_rec is defined
    list_sind is defined

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

Arguments cons [A] _ _.
Arguments nil {A}.
Check (cons 3 nil).
    cons 3 nil
         : list nat

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

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

Arguments map [A B] f l.
Arguments length {A} l.
(* A has to be maximally inserted *)

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

Arguments map [A B] f l, [A] B f l, A B f l.
Check (fun l => map length l = map (list nat) nat length l).
    fun l : list (list nat) => map length l = map length l
         : list (list nat) -> Prop
```

Note: To know which are the implicit arguments of an object, use the command *Print Implicit* (see *Displaying what the implicit arguments are*).

Warning: **Argument number *num* is a trailing implicit so must be maximal.**

For instance in

```
Arguments prod _ [_].
    Toplevel input, characters 0-21:
    > Arguments prod _ [_].
    > ^^^^^^^^^^^^^^^^^^^^^
    Warning: Argument number 1 is a trailing implicit so must be maximal
```

Automatic declaration of implicit arguments

Command: **Arguments *qualid* : default implicits**

This command tells Coq to automatically detect what are the implicit arguments of a defined object.

The auto-detection is governed by flags telling if strict, contextual, or reversible-pattern implicit arguments must be considered or not (see *Controlling strict implicit arguments*, *Controlling strict implicit arguments*, *Controlling reversible-pattern implicit arguments*, and also *Controlling the insertion of implicit arguments not followed by explicit arguments*).

Variant: **Global Arguments *qualid* : default implicits**

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

Variant: **Local Arguments *qualid* : default implicits**

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

Example

```
Inductive list (A:Set) : Set :=
| nil : list A
| cons : A -> list A -> list A.
    list is defined
    list_rect is defined
    list_ind is defined
    list_rec is defined
    list_sind is defined

Arguments cons : default implicits.
Print Implicit cons.
    cons : forall A : Set, A -> list A -> list A

    Argument A is implicit

Arguments nil : default implicits.
Print Implicit nil.
    nil : forall A : Set, list A

Set Contextual Implicit.
Arguments nil : default implicits.
Print Implicit nil.
```

(continues on next page)

(continued from previous page)

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

    Argument A is implicit and maximally inserted
```

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

```
Parameter X : Type.
    X is declared

Definition Relation := X -> X -> Prop.
    Relation is defined

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

Parameters (R : Relation) (p : Transitivity R).
    R is declared
    p is declared

Arguments p : default implicits.
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 p [x y] _ [z]

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

Parameters (a b c : X) (r1 : R a b) (r2 : R b c).
    a is declared
    b is declared
    c is declared
    r1 is declared
    r2 is declared

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

Mode for automatic declaration of implicit arguments

Flag: `Implicit Arguments`

This flag (off by default) allows to systematically declare implicit the arguments detectable as such. Auto-detection of implicit arguments is governed by flags controlling whether strict and contextual implicit arguments have to be considered or not.

Controlling strict implicit arguments

Flag: `Strict Implicit`

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

Flag: `Strongly Strict Implicit`

Use this flag (off by default) to capture exactly the strict implicit arguments and no more than the strict implicit arguments.

Controlling contextual implicit arguments

Flag: `Contextual Implicit`

By default, Coq does not automatically set implicit the contextual implicit arguments. You can turn this flag on to tell Coq to also infer contextual implicit argument.

Controlling reversible-pattern implicit arguments

Flag: `Reversible Pattern Implicit`

By default, Coq does not automatically set implicit the reversible-pattern implicit arguments. You can turn this flag on to tell Coq to also infer reversible-pattern implicit argument.

Controlling the insertion of implicit arguments not followed by explicit arguments

Flag: `Maximal Implicit Insertion`

Assuming the implicit argument mode is on, this flag (off by default) declares implicit arguments to be automatically inserted when a function is partially applied and the next argument of the function is an implicit one.

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+`. This syntax extension is given in the following grammar:

```
term       ::=   @ qualid term … term
                 @ qualid
                 qualid argument … argument
argument   ::=   term
                 (ident := term)
```

Syntax for explicitly giving implicit arguments

Example: (continued)

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

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

Renaming implicit arguments

Command: `Arguments` *qualid* *name** `: rename`

This command is used to redefine the names of implicit arguments.

Command: `Arguments` *qualid* *name** `: assert`

This command is used to assert that a given object has the expected number of arguments and that these arguments are named as expected.

Example: (continued)

```
Arguments p [s t] _ [u] _: rename.
Check (p r1 (u:=c)).
    p r1 (u:=c)
         : R b c -> R a c

Check (p (s:=a) (t:=b) r1 (u:=c) r2).
    p r1 r2
         : R a c

Fail Arguments p [s t] _ [w] _ : assert.
    The command has indeed failed with message:
    Flag "rename" expected to rename u into w.
```

Displaying what the implicit arguments are

Command: `Print Implicit` *qualid*

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

Explicit displaying of implicit arguments for pretty-printing

Flag: `Printing Implicit`

By default, the basic pretty-printing rules hide the inferable implicit arguments of an application. Turn this flag on to force printing all implicit arguments.

Flag: `Printing Implicit Defensive`

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

See also:

Printing All.

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

```
Fail Check nat = Prop.
    The command has indeed failed with message:
    The term "Prop" has type "Type" while it is expected to have type
    "Set" (universe inconsistency).
```

but succeeds in

```
Check Prop = nat.
    Prop = nat
         : Prop
```

Deactivation of implicit arguments for parsing

Flag: `Parsing Explicit`

Turning this flag on (it is off by default) deactivates the use of implicit arguments.

In this case, all arguments of constants, inductive types, constructors, etc, including the arguments declared as implicit, have to be given as if no arguments were implicit. By symmetry, this also affects printing.

Canonical structures

A canonical structure is an instance of a record/structure type that can be used to solve unification problems involving a projection applied to an unknown structure instance (an implicit argument) and a value. The complete documentation of canonical structures can be found in *Canonical Structures*; here only a simple example is given.

Command: `Canonical` `Structure`? *qualid*

This command declares *qualid* as a canonical instance of a structure (a record).

Assume that *qualid* denotes an object `(Build_struct` c_1 … c_n `)` in the structure `struct` of which the fields are x_1, …, x_n. 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.

Example

Here is an example.

```
Require Import Relations.
Require Import EqNat.
Set Implicit Arguments.
Unset Strict Implicit.
Structure Setoid : Type := {Carrier :> Set; Equal : relation Carrier;
```

(continues on next page)

(continued from previous page)

```
                       Prf_equiv : equivalence Carrier Equal}.
    Setoid is defined
    Carrier is defined
    Equal is defined
    Prf_equiv is defined

Definition is_law (A B:Setoid) (f:A -> B) := forall x y:A, Equal x y -> Equal (f x) (f y).
    is_law is defined

Axiom eq_nat_equiv : equivalence nat eq_nat.
    eq_nat_equiv is declared

Definition nat_setoid : Setoid := Build_Setoid eq_nat_equiv.
    nat_setoid is defined

Canonical nat_setoid.
```

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

```
Lemma is_law_S : is_law S.
    1 subgoal

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

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

Note: To prevent a field from being involved in the inference of canonical instances, its declaration can be annotated with the `#[canonical(false)]` attribute.

Example

For instance, when declaring the `Setoid` structure above, the `Prf_equiv` field declaration could be written as follows.

```
#[canonical(false)] Prf_equiv : equivalence Carrier Equal
```

See *Canonical Structures* for a more realistic example.

Variant: `Canonical` `Structure`? *ident* `: type`? `:=` *term*

This is equivalent to a regular definition of *ident* followed by the declaration `Canonical` *ident*.

Command: `Print Canonical Projections`

This displays the list of global names that are components of some canonical structure. For each of them, the canonical structure of which it is a projection is indicated.

Example

For instance, the above example gives the following output:

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

Note: The last line would not show up if the corresponding projection (namely `Prf_equiv`) were annotated as not canonical, as described above.

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).

Command: `Implicit Types` *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

```
Require Import List.
Implicit Types m n : nat.
Lemma cons_inj_nat : forall m n l, n :: l = m :: l -> n = m.
    1 subgoal

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

Proof.
intros m n.
    1 subgoal

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

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

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

Abort.
```

Variant: `Implicit Type` *ident* `:` *type*

This is useful for declaring the implicit type of a single variable.

Variant: `Implicit Types ( ident+ : type )+`

Adds blocks of implicit types with different specifications.

Implicit generalization

Implicit generalization is an automatic elaboration of a statement with free variables into a closed statement where these variables are quantified explicitly.

It is activated for a binder by prefixing a ‘, and for terms by surrounding it with ‘{ } or ‘( ).

Terms surrounded by ‘{ } introduce their free variables as maximally inserted implicit arguments, and terms surrounded by ‘( ) introduce them as explicit arguments.

Generalizing binders always introduce their free variables as maximally inserted implicit arguments. The binder itself introduces its argument as usual.

In the following statement, `A` and `y` are automatically generalized, `A` is implicit and `x`, `y` and the anonymous equality argument are explicit.

```
Generalizable All Variables.
Definition sym `(x:A) : `(x = y -> y = x) := fun _ p => eq_sym p.
    sym is defined

Print sym.
    sym =
    fun (A : Type) (x y : A) (p : x = y) => eq_sym p
         : forall (A : Type) (x y : A), x = y -> y = x

    Arguments sym {A}%type_scope
```

Dually to normal binders, the name is optional but the type is required:

```
Check (forall `{x = y :> A}, y = x).
    forall (A : Type) (x y : A), x = y -> y = x
         : Prop
```

When generalizing a binder whose type is a typeclass, its own class arguments are omitted from the syntax and are generalized using automatic names, without instance search. Other arguments are also generalized unless provided. This produces a fully general statement. this behaviour may be disabled by prefixing the type with a `!` or by forcing the typeclass name to be an explicit application using `@` (however the later ignores implicit argument information).

```
Class Op (A:Type) := op : A -> A -> A.
Class Commutative (A:Type) `(Op A) := commutative : forall x y, op x y = op y x.
Instance nat_op : Op nat := plus.
    nat_op is defined

Set Printing Implicit.
Check (forall `{Commutative }, True).
    forall (A : Type) (H : Op A), Commutative A H -> True
         : Prop

Check (forall `{Commutative nat}, True).
    forall H : Op nat, Commutative nat H -> True
         : Prop
```

(continues on next page)

(continued from previous page)

```
Fail Check (forall `{Commutative nat _}, True).
    The command has indeed failed with message:
    Typeclass does not expect more arguments

Fail Check (forall `{!Commutative nat}, True).
    The command has indeed failed with message:
    The term "Commutative nat" has type "Op nat -> Prop"
    which should be Set, Prop or Type.

Arguments Commutative _ {_}.
Check (forall `{!Commutative nat}, True).
    @Commutative nat nat_op -> True
         : Prop

Check (forall `{@Commutative nat plus}, True).
    @Commutative nat Nat.add -> True
         : Prop
```

Multiple binders can be merged using `,` as a separator:

```
Check (forall `{Commutative A, Hnat : !Commutative nat}, True).
    forall (A : Type) (H : Op A),
    @Commutative A H -> @Commutative nat nat_op -> True
         : Prop
```

One can control the set of generalizable identifiers with the `Generalizable` vernacular command to avoid unexpected generalizations when mistyping identifiers. There are several commands that specify which variables should be generalizable.

Command: `Generalizable All Variables`

All variables are candidate for generalization if they appear free in the context under a generalization delimiter. This may result in confusing errors in case of typos. In such cases, the context will probably contain some unexpected generalized variable.

Command: `Generalizable No Variables`

Disable implicit generalization entirely. This is the default behavior.

Command: `Generalizable` `Variable` | `Variables` *ident*+

Allow generalization of the given identifiers only. Calling this command multiple times adds to the allowed identifiers.

Command: `Global Generalizable`

Allows exporting the choice of generalizable variables.

4.2.8 Coercions

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

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

More details and examples, and a description of the commands related to coercions are provided in *Implicit Coercions*.

4.2.9 Printing constructions in full

Flag: `Printing All`

Coercions, implicit arguments, the type of pattern matching, but also notations (see *Syntax extensions and interpretation scopes*) can obfuscate the behavior of some tactics (typically the tactics applying to occurrences of subterms are sensitive to the implicit arguments). Turning this flag on 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, *`Printing All`* includes the effects of the flags *`Printing Implicit`*, *`Printing Coercions`*, *`Printing Synth`*, *`Printing Projections`*, and *`Printing Notations`*. To reactivate the high-level printing features, use the command `Unset Printing All`.

4.2.10 Printing universes

Flag: `Printing Universes`

Turn this flag on to activate the display of the actual level of each occurrence of `Type`. See *Sorts* for details. This wizard flag, in combination with *`Printing All`* can help to diagnose failures to unify terms apparently identical but internally different in the Calculus of Inductive Constructions.

Command: `Print Sorted? Universes`

This command can be used to print the constraints on the internal level of the occurrences of `Type` (see *Sorts*).

If the `Sorted` keyword is present, each universe will be made equivalent to a numbered label reflecting its level (with a linear ordering) in the universe hierarchy.

Variant: `Print Sorted? Universes` *`string`*

This variant accepts an optional output filename.

If *`string`* ends in `.dot` or `.gv`, the constraints are printed in the DOT language, and can be processed by Graphviz tools. The format is unspecified if `string` doesn't end in `.dot` or `.gv`.

Variant: `Print Universes Subgraph(`*`qualid`*`+)`

Prints the graph restricted to the requested names (adjusting constraints to preserve the implied transitive constraints between kept universes).

4.2.11 Existential variables

Coq terms can include existential variables which represents unknown subterms to eventually be replaced by actual subterms.

Existential variables are generated in place of unsolvable implicit arguments or "_" placeholders when using commands such as `Check` (see Section *Requests to the environment*) or when using tactics such as *`refine`*, as well as in place of unsolvable instances when using tactics such that *`eapply`*. An existential variable is defined in a context, which is the context of variables of the placeholder which generated the existential variable, and a type, which is the expected type of the placeholder.

As a consequence of typing constraints, existential variables can be duplicated in such a way that they possibly appear in different contexts than their defining context. Thus, any occurrence of a given existential variable comes with an instance of its original context. In the simple case, when an existential variable denotes the placeholder which generated it, or is used in the same context as the one in which it was generated, the context is not displayed and the existential variable is represented by "?" followed by an identifier.

```
Parameter identity : forall (X:Set), X -> X.
    identity is declared

Check identity _ _.
    identity ?X ?y
         : ?X
    where
    ?X : [ |- Set]
    ?y : [ |- ?X]

Check identity _ (fun x => _).
    identity (forall x : ?S, ?S0) (fun x : ?S => ?y)
         : forall x : ?S, ?S0
    where
    ?S : [ |- Set]
    ?S0 : [x : ?S |- Set]
    ?y : [x : ?S |- ?S0]
```

In the general case, when an existential variable `?`*`ident`* appears outside of its context of definition, its instance, written under the form `{` *`ident`* `:=` *`term`*$^{*}_{;}$ `}` is appending to its name, indicating how the variables of its defining context are instantiated. The variables of the context of the existential variables which are instantiated by themselves are not written, unless the *`Printing Existential Instances`* flag is on (see Section *Explicit displaying of existential instances for pretty-printing*), and this is why an existential variable used in the same context as its context of definition is written with no instance.

```
Check (fun x y => _) 0 1.
    (fun x y : nat => ?y) 0 1
         : ?T@{x:=0; y:=1}
    where
    ?T : [x : nat  y : nat |- Type]
    ?y : [x : nat  y : nat |- ?T]

Set Printing Existential Instances.
Check (fun x y => _) 0 1.
    (fun x y : nat => ?y@{x:=x; y:=y}) 0 1
         : ?T@{x:=0; y:=1}
    where
    ?T : [x : nat  y : nat |- Type]
    ?y : [x : nat  y : nat |- ?T@{x:=x; y:=y}]
```

Existential variables can be named by the user upon creation using the syntax `?[`*`ident`*`]`. This is useful when the existential variable needs to be explicitly handled later in the script (e.g. with a named-goal selector, see *Goal selectors*).

Explicit displaying of existential instances for pretty-printing

Flag: `Printing Existential Instances`

This flag (off by default) activates the full display of how the context of an existential variable is instantiated at each of the occurrences of the existential variable.

Solving existential variables using tactics

Instead of letting the unification engine try to solve an existential variable by itself, one can also provide an explicit hole together with a tactic to solve it. Using the syntax `ltac:(tacexpr)`, the user can put a tactic anywhere a term is expected. The order of resolution is not specified and is implementation-dependent. The inner tactic may use any variable defined in its scope, including repeated alternations between variables introduced by term binding as well as those introduced by tactic binding. The expression `tacexpr` can be any tactic expression as described in *Ltac*.

```
Definition foo (x : nat) : nat := ltac:(exact x).
    foo is defined
```

This construction is useful when one wants to define complicated terms using highly automated tactics without resorting to writing the proof-term by means of the interactive proof engine.

4.2.12 Primitive Integers

The language of terms features 63-bit machine integers as values. The type of such a value is *axiomatized*; it is declared through the following sentence (excerpt from the `Int63` module):

```
Primitive int := #int63_type.
```

This type is equipped with a few operators, that must be similarly declared. For instance, equality of two primitive integers can be decided using the `Int63.eqb` function, declared and specified as follows:

```
Primitive eqb := #int63_eq.
Notation "m '==' n" := (eqb m n) (at level 70, no associativity) : int63_scope.

Axiom eqb_correct : forall i j, (i == j)%int63 = true -> i = j.
```

The complete set of such operators can be obtained looking at the `Int63` module.

These primitive declarations are regular axioms. As such, they must be trusted and are listed by the `Print Assumptions` command, as in the following example.

```
From Coq Require Import Int63.
Lemma one_minus_one_is_zero : (1 - 1 = 0)%int63.
Proof.
apply eqb_correct; vm_compute; reflexivity.
Qed.


Print Assumptions one_minus_one_is_zero.
    Axioms:
    sub : int -> int -> int
    eqb_correct : forall i j : int, (i == j)%int63 = true -> i = j
    eqb : int -> int -> bool
```

The reduction machines (*`vm_compute`*, *`native_compute`*) implement dedicated, efficient, rules to reduce the applications of these primitive operations.

The extraction of these primitives can be customized similarly to the extraction of regular axioms (see *Extraction of programs in OCaml and Haskell*). Nonetheless, the `ExtrOCamlInt63` module can be used when extracting to OCaml: it maps the Coq primitives to types and functions of a `Uint63` module. Said OCaml module is not produced by extraction. Instead, it has to be provided by the user (if they want to compile or execute the extracted code). For instance, an implementation of this module can be taken from the kernel of Coq.

Literal values (at type Int63.int) are extracted to literal OCaml values wrapped into the Uint63.of_int (resp. Uint63.of_int64) constructor on 64-bit (resp. 32-bit) platforms. Currently, this cannot be customized (see the function Uint63.compile from the kernel).

4.2.13 Primitive Floats

The language of terms features Binary64 floating-point numbers as values. The type of such a value is *axiomatized*; it is declared through the following sentence (excerpt from the PrimFloat module):

```
Primitive float := #float64_type.
```

This type is equipped with a few operators, that must be similarly declared. For instance, the product of two primitive floats can be computed using the PrimFloat.mul function, declared and specified as follows:

```
Primitive mul := #float64_mul.
Notation "x * y" := (mul x y) : float_scope.

Axiom mul_spec : forall x y, Prim2SF (x * y)%float = SF64mul (Prim2SF x) (Prim2SF y).
```

where Prim2SF is defined in the FloatOps module.

The set of such operators is described in section *Floats library*.

These primitive declarations are regular axioms. As such, they must be trusted, and are listed by the Print Assumptions command.

The reduction machines (*vm_compute*, *native_compute*) implement dedicated, efficient rules to reduce the applications of these primitive operations, using the floating-point processor operators that are assumed to comply with the IEEE 754 standard for floating-point arithmetic.

The extraction of these primitives can be customized similarly to the extraction of regular axioms (see *Extraction of programs in OCaml and Haskell*). Nonetheless, the ExtrOCamlFloats module can be used when extracting to OCaml: it maps the Coq primitives to types and functions of a Float64 module. Said OCaml module is not produced by extraction. Instead, it has to be provided by the user (if they want to compile or execute the extracted code). For instance, an implementation of this module can be taken from the kernel of Coq.

Literal values (of type Float64.t) are extracted to literal OCaml values (of type float) written in hexadecimal notation and wrapped into the Float64.of_float constructor, e.g.: Float64.of_float (0x1p+0).

4.2.14 Bidirectionality hints

When type-checking an application, Coq normally does not use information from the context to infer the types of the arguments. It only checks after the fact that the type inferred for the application is coherent with the expected type. Bidirectionality hints make it possible to specify that after type-checking the first arguments of an application, typing information should be propagated from the context to help inferring the types of the remaining arguments.

Command: Arguments *qualid* *ident*$_1$* & *ident*$_2$*

This commands tells the typechecking algorithm, when type-checking applications of *qualid*, to first type-check the arguments in *ident*$_1$ and then propagate information from the typing context to type-check the remaining arguments (in *ident*$_2$).

Example

In a context where a coercion was declared from bool to nat:

```
Definition b2n (b : bool) := if b then 1 else 0.
Coercion b2n : bool >-> nat.
```

Coq cannot automatically coerce existential statements over `bool` to statements over `nat`, because the need for inserting a coercion is known only from the expected type of a subterm:

```
Fail Check (ex_intro _ true _ : exists n : nat, n > 0).
    The command has indeed failed with message:
    The term "ex_intro ?P true ?y" has type "exists y, ?P y"
    while it is expected to have type "exists n : nat, n > 0"
    (cannot unify "bool" and "nat").
```

However, a suitable bidirectionality hint makes the example work:

```
Arguments ex_intro _ _ & _ _.
Check (ex_intro _ true _ : exists n : nat, n > 0).
    ex_intro (fun n : nat => n > 0) true ?g : exists n : nat, n > 0
         : exists n : nat, n > 0
    where
    ?g : [ |- (fun n : nat => n > 0) true]
```

Coq will attempt to produce a term which uses the arguments you provided, but in some cases involving Program mode the arguments after the bidirectionality starts may be replaced by convertible but syntactically different terms.

4.3 The Coq library

The Coq library has two parts:

- **The basic library**: definitions and theorems for the most commonly used elementary logical notions and data types. Coq normally loads these files automatically when it starts.
- **The standard library**: general-purpose libraries with definitions and theorems for sets, lists, sorting, arithmetic, etc. To use these files, users must load them explicitly with the `Require` command (see *Compiled files*)

There are also many libraries provided by Coq users' community. These libraries and developments are available for download at http://coq.inria.fr (see *Users' contributions*).

This chapter briefly reviews the Coq libraries whose contents can also be browsed at http://coq.inria.fr/stdlib/.

4.3.1 The basic library

This section lists the basic notions and results which are directly available in the standard Coq system. Most of these constructions are defined in the `Prelude` module in directory `theories/Init` at the Coq root directory; this includes the modules `Notations`, `Logic`, `Datatypes`, `Specif`, `Peano`, `Wf` and `Tactics`. Module `Logic_Type` also makes it in the initial state.

Notations

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

Notation	Precedence	Associativity
_ -> _	99	right
_ <-> _	95	no
_ \/ _	85	right
_ /\ _	80	right
~ _	75	right
_ = _	70	no
_ = _ = _	70	no
_ = _ :> _	70	no
_ <> _	70	no
_ <> _ :> _	70	no
_ < _	70	no
_ > _	70	no
_ <= _	70	no
_ >= _	70	no
_ < _ < _	70	no
_ < _ <= _	70	no
_ <= _ < _	70	no
_ <= _ <= _	70	no
_ + _	50	left
_ || _	50	left
_ - _	50	left
_ * _	40	left
_ _	40	left
_ / _	40	left
- _	35	right
/ _	35	right
_ ^ _	30	right

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 of *form* is shown below:

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

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

Propositional Connectives

First, we find propositional calculus connectives:

```
Inductive True : Prop := I.
Inductive False :  Prop := .
Definition not (A: Prop) := A -> False.
Inductive and (A B:Prop) : Prop := conj (_:A) (_:B).
Section Projections.
 Variables A B : Prop.
 Theorem proj1 : A /\ B -> A.
 Theorem proj2 : A /\ B -> B.
End Projections.
Inductive or (A B:Prop) : Prop :=
| or_introl (_:A)
| or_intror (_:B).
Definition iff (P Q:Prop) := (P -> Q) /\ (Q -> P).
Definition IF_then_else (P Q R:Prop) := P /\ Q \/ ~ P /\ R.
```

Quantifiers

Then we find first-order quantifiers:

```
Definition all (A:Set) (P:A -> Prop) := forall x:A, P x.
Inductive ex (A: Set) (P:A -> Prop) : Prop :=
 ex_intro (x:A) (_:P x).
Inductive ex2 (A:Set) (P Q:A -> Prop) : Prop :=
 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.

```
Inductive eq (A:Type) (x:A) : A -> Prop :=
  eq_refl : eq A x x.
```

Lemmas

Finally, a few easy lemmas are provided.

```
Theorem absurd : forall A C:Prop, A -> ~ A -> C.
Section equality.
Variables A B : Type.
Variable f : A -> B.
Variables x y z : A.
Theorem eq_sym : x = y -> y = x.
Theorem eq_trans : x = y -> y = z -> x = z.
Theorem f_equal : x = y -> f x = f y.
Theorem not_eq_sym : x <> y -> y <> x.
End equality.
Definition eq_ind_r :
 forall (A:Type) (x:A) (P:A->Prop), P x -> forall y:A, y = x -> P y.
Definition eq_rec_r :
 forall (A:Type) (x:A) (P:A->Set), P x -> forall y:A, y = x -> P y.
Definition eq_rect_r :
 forall (A:Type) (x:A) (P:A->Type), P x -> forall y:A, y = x -> P y.
Hint Immediate eq_sym not_eq_sym : 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.

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

Datatypes

In the basic library, we find in `Datatypes.v` the definition of the basic data-types of programming, defined as inductive constructions over the sort `Set`. Some of them come with a special syntax shown below (this syntax table is common with the next section *Specification*):

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

Programming

```
Inductive unit : Set := tt.
Inductive bool : Set := true | false.
Inductive nat : Set := O | S (n:nat).
```

(continues on next page)

(continued from previous page)

```
Inductive option (A:Set) : Set := Some (_:A) | None.
Inductive identity (A:Type) (a:A) : A -> Type :=
  refl_identity : identity A a a.
```

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

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

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

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

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

Specification

The following notions defined in module `Specif.v` allow to build new data-types and specifications. They are available with the syntax shown in the previous section *Datatypes*.

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

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

```
Inductive sig (A:Set) (P:A -> Prop) : Set := exist (x:A) (_:P x).
Inductive sig2 (A:Set) (P Q:A -> Prop) : Set :=
  exist2 (x:A) (_:P x) (_:Q x).
```

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

```
Inductive sigT (A:Type) (P:A -> Type) : Type := existT (x:A) (_:P x).
Section Projections2.
Variable A : Type.
Variable P : A -> Type.
Definition projT1 (H:sigT A P) := let (x, h) := H in x.
Definition projT2 (H:sigT A P) :=
 match H return P (projT1 H) with
  existT _ _ x h => h
 end.
End Projections2.
```

(continues on next page)

(continued from previous page)

```
Inductive sigT2 (A: Type) (P Q:A -> Type) : Type :=
  existT2 (x:A) (_:P x) (_:Q x).
```

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

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

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

```
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.

```
Lemma Choice :
 forall (S S':Set) (R:S -> S' -> Prop),
  (forall x:S, {y : S' | R x y}) ->
  {f : S -> S' | forall z:S, R z (f z)}.
Lemma Choice2 :
 forall (S S':Set) (R:S -> S' -> Set),
  (forall x:S, {y : S' &  R x y}) ->
   {f : S -> S' &  forall z:S, R z (f z)}.
Lemma bool_choice :
 forall (S:Set) (R1 R2:S -> Prop),
  (forall x:S, {R1 x} + {R2 x}) ->
  {f : S -> bool |
   forall x:S, f x = true /\ R1 x \/ f x = false /\ R2 x}.
```

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

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

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

```
Definition except := False_rec.
Theorem absurd_set : forall (A:Prop) (C:Set), A -> ~ A -> C.
Theorem and_rect2 :
 forall (A B:Prop) (P:Type), (A -> B -> P) -> A /\ B -> P.
```

Basic Arithmetic

The basic library includes a few elementary properties of natural numbers, together with the definitions of predecessor, addition and multiplication, in module `Peano.v`. It also provides a scope `nat_scope` gathering standard notations for common operations (`+`, `*`) and a decimal notation for numbers, allowing for instance to write `3` for `S (S (S O)))`. This also works on the left hand side of a `match` expression (see for example section *refine*). This scope is opened by default.

Example

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

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

Now comes the content of module Peano:

```
Theorem eq_S : forall x y:nat, x = y -> S x = S y.
Definition pred (n:nat) : nat :=
 match n with
 | 0 => 0
 | S u => u
 end.
Theorem pred_Sn : forall m:nat, m = pred (S m).
Theorem eq_add_S : forall n m:nat, S n = S m -> n = m.
Hint Immediate eq_add_S : core.
Theorem not_eq_S : forall n m:nat, n <> m -> S n <> S m.
Definition IsSucc (n:nat) : Prop :=
 match n with
 | 0 => False
 | S p => True
 end.
Theorem O_S : forall n:nat, 0 <> S n.
Theorem n_Sn : forall n:nat, n <> S n.
Fixpoint plus (n m:nat) {struct n} : nat :=
 match n with
 | 0 => m
 | S p => S (p + m)
 end
where "n + m" := (plus n m) : nat_scope.
Lemma plus_n_O : forall n:nat, n = n + 0.
Lemma plus_n_Sm : forall n m:nat, S (n + m) = n + S m.
Fixpoint mult (n m:nat) {struct n} : nat :=
 match n with
 | 0 => 0
 | S p => m + p * m
 end
where "n * m" := (mult n m) : nat_scope.
Lemma mult_n_O : forall n:nat, 0 = n * 0.
Lemma mult_n_Sm : forall n m:nat, n * m + n = n * (S m).
```

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

```
Inductive le (n:nat) : nat -> Prop :=
| le_n : le n n
| le_S : forall m:nat, n <= m -> n <= (S m)
where "n <= m" := (le n m) : nat_scope.
Definition lt (n m:nat) := S n <= m.
Definition ge (n m:nat) := m <= n.
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.

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

Well-founded recursion

The basic library contains the basics of well-founded recursion and well-founded induction, in module `Wf.v`.

```
Section Well_founded.
Variable A : Type.
Variable R : A -> A -> Prop.
Inductive Acc (x:A) : Prop :=
  Acc_intro : (forall y:A, R y x -> Acc y) -> Acc x.
Lemma Acc_inv x : Acc x -> forall y:A, R y x -> Acc y.
Definition well_founded := forall a:A, Acc a.
Hypothesis Rwf : well_founded.
Theorem well_founded_induction :
 forall P:A -> Set,
  (forall x:A, (forall y:A, R y x -> P y) -> P x) -> forall a:A, P a.
Theorem well_founded_ind :
 forall P:A -> Prop,
  (forall x:A, (forall y:A, R y x -> P y) -> P x) -> forall a:A, P a.
```

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

```
Section FixPoint.
Variable P : A -> Type.
Variable F : forall x:A, (forall y:A, R y x -> P y) -> P x.
Fixpoint Fix_F (x:A) (r:Acc x) {struct r} : P x :=
  F x (fun (y:A) (p:R y x) => Fix_F y (Acc_inv x r y p)).
Definition Fix (x:A) := Fix_F x (Rwf x).
Hypothesis F_ext :
  forall (x:A) (f g:forall y:A, R y x -> P y),
    (forall (y:A) (p:R y x), f y p = g y p) -> F x f = F x g.
Lemma Fix_F_eq :
 forall (x:A) (r:Acc x),
   F x (fun (y:A) (p:R y x) => Fix_F y (Acc_inv x r y p)) = Fix_F x r.
Lemma Fix_F_inv : forall (x:A) (r s:Acc x), Fix_F x r = Fix_F x s.
Lemma fix_eq : forall x:A, Fix x = F x (fun (y:A) (p:R y x) => Fix y).
End FixPoint.
End Well_founded.
```

Accessing the Type level

The standard library includes `Type` level definitions of counterparts of some logic concepts and basic lemmas about them.

The module `Datatypes` defines `identity`, which is the `Type` level counterpart of equality:

```
Inductive identity (A:Type) (a:A) : A -> Type :=
  identity_refl : identity A a a.
```

Some properties of identity are proved in the module Logic_Type, which also provides the definition of Type level negation:

```
Definition notT (A:Type) := A -> False.
```

Tactics

A few tactics defined at the user level are provided in the initial state, in module Tactics.v. They are listed at http://coq.inria.fr/stdlib, in paragraph Init, link Tactics.

4.3.2 The standard library

Survey

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

- **Logic** : Classical logic and dependent equality
- **Arith** : Basic Peano arithmetic
- **PArith** : Basic positive integer arithmetic
- **NArith** : Basic binary natural number arithmetic
- **ZArith** : Basic relative integer arithmetic
- **Numbers** : Various approaches to natural, integer and cyclic numbers (currently axiomatically and on top of 2^31 binary words)
- **Bool** : Booleans (basic functions and results)
- **Lists** : Monomorphic and polymorphic lists (basic functions and results), Streams (infinite sequences defined with co-inductive types)
- **Sets** : Sets (classical, constructive, finite, infinite, power set, etc.)
- **FSets** : Specification and implementations of finite sets and finite maps (by lists and by AVL trees)
- **Reals** : Axiomatization of real numbers (classical, basic functions, integer part, fractional part, limit, derivative, Cauchy series, power series and results,...)
- **Floats** : Machine implementation of floating-point arithmetic (for the binary64 format)
- **Relations** : Relations (definitions and basic results)
- **Sorting** : Sorted list (basic definitions and heapsort correctness)
- **Strings** : 8-bits characters and strings
- **Wellfounded** : Well-founded relations (basic results)

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

The different modules of the Coq standard library are documented online at https://coq.inria.fr/stdlib.

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.

The following table describes the notations available in scope nat_scope :

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

Notations for integer arithmetic

The following table describes the syntax of expressions for integer arithmetic. It is provided by requiring and opening the module ZArith and opening scope Z_scope. It specifies how notations are interpreted and, when not already reserved, the precedence and associativity.

Notation	Interpretation	Precedence	Associativity
_ < _	Z.lt		
_ <= _	Z.le		
_ > _	Z.gt		
_ >= _	Z.ge		
x < y < z	x < y /\ y < z		
x < y <= z	x < y /\ y <= z		
x <= y < z	x <= y /\ y < z		
x <= y <= z	x <= y /\ y <= z		
_ ?= _	Z.compare	70	no
_ + _	Z.add		
_ - _	Z.sub		
_ * _	Z.mul		
_ / _	Z.div		
_ mod _	Z.modulo	40	no
- _	Z.opp		
_ ^ _	Z.pow		

Example

```
Require Import ZArith.
    [Loading ML file newring_plugin.cmxs ... done]
    [Loading ML file zify_plugin.cmxs ... done]
```

(continues on next page)

(continued from previous page)

```
    [Loading ML file omega_plugin.cmxs ... done]

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

Open Scope Z_scope.
Check 2 + 3.
    2 + 3
         : Z
```

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 arithmetic. The inverse function was added.

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

Example

```
Require Import Reals.
    [Loading ML file r_syntax_plugin.cmxs ... done]
    [Loading ML file micromega_plugin.cmxs ... done]

Check  (2 + 3)%R.
    (2 + 3)%R
         : R

Open Scope R_scope.
Check 2 + 3.
    2 + 3
         : R
```

Some tactics for real numbers

In addition to the powerful `ring`, `field` and `lra` tactics (see Chapter *Tactics*), there are also:

`discrR`
Proves that two real integer constants are different.

Example

```
Require Import DiscrR.
Open Scope R_scope.
Goal 5 <> 0.
    1 subgoal

      ============================
      5 <> 0

discrR.
```

`split_Rabs`
Allows unfolding the `Rabs` constant and splits corresponding conjunctions.

Example

```
Require Import Reals.
Open Scope R_scope.
Goal forall x:R, x <= Rabs x.
    1 subgoal

      ============================
      forall x : R, x <= Rabs x

intro; split_Rabs.
    2 subgoals

      x : R
      Hlt : x < 0
      ============================
      x <= - x

    subgoal 2 is:
     x <= x
```

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

Example

```
Require Import Reals.
Open Scope R_scope.
Goal forall x y z:R, x * y * z <> 0.
    1 subgoal
```

(continues on next page)

(continued from previous page)

```
      ============================
      forall x y z : R, x * y * z <> 0

intros; split_Rmult.
    3 subgoals

      x, y, z : R
      ============================
      x <> 0

    subgoal 2 is:
     y <> 0
    subgoal 3 is:
     z <> 0
```

These tactics has been written with the tactic language $\mathtt{L}_{\text{tac}}$ described in Chapter *Ltac*.

List library

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

It defines the following notions:

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

The following table shows notations available when opening scope `list_scope`.

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

Floats library

The library of primitive floating-point arithmetic can be loaded by requiring module `Floats`:

```
Require Import Floats.
```

It exports the module `PrimFloat` that provides a primitive type named `float`, defined in the kernel (see section *Primitive Floats*), as well as two variant types `float_comparison` and `float_class`:

```
Print float.
    *** [ float : Set ]

Print float_comparison.
    Variant float_comparison : Set :=
        FEq : float_comparison
      | FLt : float_comparison
      | FGt : float_comparison
      | FNotComparable : float_comparison

Print float_class.
    Variant float_class : Set :=
        PNormal : float_class
      | NNormal : float_class
      | PSubn : float_class
      | NSubn : float_class
      | PZero : float_class
      | NZero : float_class
      | PInf : float_class
      | NInf : float_class
      | NaN : float_class
```

It then defines the primitive operators below, using the processor floating-point operators for binary64 in rounding-to-nearest even:

- `abs`
- `opp`
- `sub`
- `add`
- `mul`
- `div`
- `sqrt`
- `compare` : compare two floats and return a `float_comparison`
- `classify` : analyze a float and return a `float_class`
- `of_int63` : round a primitive integer and convert it into a float
- `normfr_mantissa` : take a float in `[0.5; 1.0)` and return its mantissa
- `frshiftexp` : convert a float to fractional part in `[0.5; 1.0)` and integer part
- `ldshiftexp` : multiply a float by an integral power of 2
- `next_up` : return the next float towards positive infinity
- `next_down` : return the next float towards negative infinity

For special floating-point values, the following constants are also defined:

- `zero`
- `neg_zero`
- `one`

- `two`
- `infinity`
- `neg_infinity`
- `nan` : Not a Number (assumed to be unique: the ”payload” of NaNs is ignored)

The following table shows the notations available when opening scope `float_scope`.

Notation	Interpretation
`- _`	`opp`
`_ - _`	`sub`
`_ + _`	`add`
`_ * _`	`mul`
`_ / _`	`div`
`_ == _`	`eqb`
`_ < _`	`ltb`
`_ <= _`	`leb`
`_ ?= _`	`compare`

Floating-point constants are parsed and pretty-printed as (17-digit) decimal constants. This ensures that the composition parse ∘ print amounts to the identity.

Warning: `The constant` *`numeral`* `is not a binary64 floating-point value. A closest value will be used and`

Not all decimal constants are floating-point values. This warning is generated when parsing such a constant (for instance `0.1`).

Example

```
Open Scope float_scope.
Eval compute in 1 + 0.5.
    = 1.5
         : float

Eval compute in 1 / 0.
    = infinity
         : float

Eval compute in 1 / -0.
    = neg_infinity
         : float

Eval compute in 0 / 0.
    = nan
         : float

Eval compute in 0 ?= -0.
    = FEq
         : float_comparison

Eval compute in nan ?= nan.
    = FNotComparable
         : float_comparison

Eval compute in next_down (-1).
```

(continues on next page)

(continued from previous page)

```
    = -1.0000000000000002
         : float
```

The primitive operators are specified with respect to their Gallina counterpart, using the variant type `spec_float`, and the injection `Prim2SF`:

```
Print spec_float.
    Variant spec_float : Set :=
        S754_zero : bool -> spec_float
      | S754_infinity : bool -> spec_float
      | S754_nan : spec_float
      | S754_finite : bool -> positive -> Z -> spec_float

    Arguments S754_zero _%bool_scope
    Arguments S754_infinity _%bool_scope
    Arguments S754_finite _%bool_scope _%positive_scope _%Z_scope

Check Prim2SF.
    Prim2SF
         : float -> spec_float

Check mul_spec.
    mul_spec
         : forall x y : float, Prim2SF (x * y) = SF64mul (Prim2SF x) (Prim2SF y)
```

For more details on the available definitions and lemmas, see the online documentation of the `Floats` library.

4.3.3 Users' contributions

Numerous users' contributions have been collected and are available at URL http://coq.inria.fr/opam/www/. 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. You will also find informations on how to submit a new contribution.

4.4 Calculus of Inductive Constructions

The underlying formal language of Coq is a *Calculus of Inductive Constructions* (Cic) whose inference rules are presented in this chapter. The history of this formalism as well as pointers to related work are provided in a separate chapter; see *Credits*.

4.4.1 The terms

The expressions of the Cic are *terms* and all terms 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, universal quantification is relative to a type and takes the form "*for all* x *of type* T, P". The expression "x *of type* T" is written "$x : T$". Informally, "$x : T$" can be thought as "x *belongs to* T".

The types of types are *sorts*. Types and sorts are themselves terms so that terms, types and sorts are all components of a common syntactic language of terms which is described in Section *Terms* but, first, we describe sorts.

Sorts

All sorts have a type and there is an infinite well-founded typing hierarchy of sorts whose base sorts are SProp, Prop and Set.

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

The sort SProp is like Prop but the propositions in SProp are known to have irrelevant proofs (all proofs are equal). Objects of type SProp are called strict propositions. See *SProp (proof irrelevant propositions)* for information about using SProp, and *[GCST19]* for meta theoretical considerations.

The sort Set intends to be the type of small sets. This includes data types such as booleans and naturals, but also products, subsets, and function types over these data types.

SProp, Prop and Set themselves can be manipulated as ordinary terms. Consequently they also have a type. Because assuming simply that Set has type Set leads to an inconsistent theory *[Coq86]*, the language of Cic has infinitely many sorts. There are, in addition to the base sorts, a hierarchy of universes $\mathsf{Type}(i)$ for any integer $i \geq 1$.

Like Set, all of the sorts $\mathsf{Type}(i)$ contain small sets such as booleans, natural numbers, as well as products, subsets and function types over small sets. But, unlike Set, they also contain large sets, namely the sorts Set and $\mathsf{Type}(j)$ for $j < i$, and all products, subsets and function types over these sorts.

Formally, we call $\mathcal{S}$ the set of sorts which is defined by:

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

Their properties, such as: $\mathsf{Prop} : \mathsf{Type}(1)$, $\mathsf{Set} : \mathsf{Type}(1)$, and $\mathsf{Type}(i) : \mathsf{Type}(i+1)$, are defined in Section *Subtyping rules*.

The user does not have to mention explicitly the index i when referring to the universe $\mathsf{Type}(i)$. 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 $\mathsf{Type} : \mathsf{Type}$. We shall make precise in the typing rules the constraints between the indices.

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

See also:

Section *Printing universes*.

Terms

Terms are built from sorts, variables, constants, abstractions, applications, local definitions, and products. From a syntactic point of view, types cannot be distinguished from terms, except that they cannot start by an abstraction or a constructor. More precisely the language of the *Calculus of Inductive Constructions* is built from the following rules.

1. the sorts SProp, Prop, Set, $\mathsf{Type}(i)$ are terms.
2. variables, hereafter ranged over by letters x, y, etc., are terms

3. constants, hereafter ranged over by letters c, d, etc., 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 does not occur 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 => u` in Coq concrete syntax) is a term. This is a notation for the λ-abstraction of λ-calculus *[Bar81]*. The term $\lambda x : T.\ u$ is a function which maps elements of T to the expression 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, T and u are terms then $\mathsf{let}\ x := t : T\ \mathsf{in}\ u$ is a term which denotes the term u where the variable x is locally bound to t of type T. This stands for the common "let-in" construction of functional programs such as ML or Scheme.

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.

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 $u\{x/t\}$.

The logical vs programming readings. The constructions of the Cic can be used to express both logical and programming notions, accordingly to the Curry-Howard correspondence between proofs and programs, and between propositions and types *[CFC58][How80][dB72]*.

For instance, let us assume that nat is the type of natural numbers with zero element written 0 and that `True` is the always true proposition. Then $\rightarrow$ is used both to denote $\mathsf{nat} \rightarrow \mathsf{nat}$ which is the type of functions from nat to nat, to denote True→True which is an implicative proposition, to denote $\mathsf{nat} \rightarrow \mathsf{Prop}$ which is the type of unary predicates over the natural numbers, etc.

Let us assume that `mult` is a function of type $\mathsf{nat} \rightarrow \mathsf{nat} \rightarrow \mathsf{nat}$ and `eqnat` a predicate of type $\mathsf{nat} \rightarrow \mathsf{nat} \rightarrow \mathsf{Prop}$. The λ-abstraction can serve to build "ordinary" functions as in $\lambda x : \mathsf{nat}.\ (\mathsf{mult}\ x\ x)$ (i.e. `fun x:nat => mult x x` in Coq notation) but may build also predicates over the natural numbers. For instance $\lambda x : \mathsf{nat}.\ (\mathsf{eqnat}\ x\ 0)$ (i.e. `fun x:nat => eqnat x 0` in Coq notation) will represent the predicate of one variable x which asserts the equality of x with 0. This predicate has type $\mathsf{nat} \rightarrow \mathsf{Prop}$ and it can be applied to any expression of type nat, say t, to give an object $P\ t$ of type Prop, namely a proposition.

Furthermore `forall x:nat, P x` will represent the type of functions which associate to each natural number n an object of type $(P\ n)$ and consequently represent the type of proofs of the formula "$\forall x.\ P(x)$".

4.4.2 Typing rules

As objects of type theory, terms are subjected to *type discipline*. The well typing of a term depends on a global environment and a local context.

Local context. A *local context* is an ordered list of *local declarations* of names which we call *variables*. The declaration of some variable x is either a *local assumption*, written $x : T$ (T is a type) or a *local definition*, written $x := t : T$. We use brackets to write local contexts. A typical example is $[x : T;\ y := u : U;\ z : V]$. Notice that the variables declared in a local context must be distinct. If Γ is a local context that 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 $(x := t : T) \in \Gamma$. For the rest of the chapter, $\Gamma :: (y : T)$ denotes the local context Γ enriched with the local assumption $y : T$. Similarly, $\Gamma :: (y := t : T)$ denotes the local context Γ enriched with the local definition $(y := t : T)$. The notation $[]$ denotes the empty local context. By $\Gamma_1; \Gamma_2$ we mean concatenation of the local context Γ_1 and the local context Γ_2.

Global environment. A *global environment* is an ordered list of *global declarations*. Global declarations are either *global assumptions* or *global definitions*, but also declarations of inductive objects. Inductive objects themselves declare both inductive or coinductive types and constructors (see Section *Inductive Definitions*).

A *global assumption* will be represented in the global environment as $(c : T)$ which assumes the name c to be of some type T. A *global definition* will be represented in the global environment as $c := t : T$ which defines the name c to have value t and type T. We shall call such names *constants*. For the rest of the chapter, the $E;\ c : T$ denotes the global environment E enriched with the global assumption $c : T$. Similarly, $E;\ c := t : T$ denotes the global environment E enriched with the global definition $(c := t : T)$.

The rules for inductive definitions (see Section *Inductive Definitions*) 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 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 global environment E and local context Γ. The second judgment $\mathcal{WF}(E)[\Gamma]$ means that the global environment E is well-formed and the local context Γ is a valid local context in this global environment.

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

W-Empty

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

W-Local-Assum

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

W-Local-Def

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

W-Global-Assum

$$\frac{E[] \vdash T : s \qquad s \in \mathcal{S} \qquad c \notin E}{\mathcal{WF}(E;\ c : T)[]}$$

W-Global-Def

$$\frac{E[] \vdash t : T \qquad c \notin E}{\mathcal{WF}(E;\ c := t : T)[]}$$

Ax-SProp

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{SProp} : \mathsf{Type}(1)}$$

Ax-Prop

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{Prop} : \mathsf{Type}(1)}$$

Ax-Set

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{Set} : \mathsf{Type}(1)}$$

Ax-Type

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{Type}(i) : \mathsf{Type}(i+1)}$$

Var

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

Const

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

Prod-SProp

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

Prod-Prop

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

Prod-Set

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

Prod-Type

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \{\mathsf{SProp}, \mathsf{Type}i\} \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{Type}(i)}{E[\Gamma] \vdash \forall x : T,\ U : \mathsf{Type}(i)}$$

Lam

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

App

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

Let

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

Note: **Prod-Prop** and **Prod-Set** typing-rules make sense if we consider the semantic difference between Prop and Set:

- All values of a type that has a sort Set are extractable.
- No values of a type that has a sort Prop are extractable.

Note: We may have let $x := t : T$ 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 *Conversion rules*).

4.4.3 Conversion rules

In Cic, there is an internal reduction mechanism. In particular, it can decide if two programs are *intentionally* equal (one says *convertible*). Convertibility is described in this section.

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 global environment E and local 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 [Coq85].

ι-reduction

A specific conversion rule is associated to the inductive objects in the global environment. We shall give later on (see Section *Well-formed inductive definitions*) 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 [PM93a][Wer94].

δ-reduction

We may have variables defined in local contexts or constants defined 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.

Delta-Local

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (x := t : T) \in \Gamma}{E[\Gamma] \vdash x\ \triangleright_\Delta\ t}$$

Delta-Global

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (c := t : T) \in E}{E[\Gamma] \vdash c \ \triangleright_\delta \ t}$$

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.

Zeta

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad E[\Gamma] \vdash u : U \qquad E[\Gamma :: (x := u : U)] \vdash t : T}{E[\Gamma] \vdash \mathsf{let}\ x := u : U\ \mathsf{in}\ t \ \triangleright_\zeta \ t\{x/u\}}$$

η-expansion

Another important concept is η-expansion. It is legal to identify any term t of functional type $\forall x : T,\ U$ with its so-called η-expansion

$$\lambda x : T.\ (t\ x)$$

for x an arbitrary variable name fresh in t.

Note: We deliberately do not define η-reduction:

$$\lambda x : T.\ (t\ x) \ \not\triangleright_\eta \ t$$

This is because, in general, the type of t need not to be convertible to the type of $\lambda x : T.\ (t\ x)$. E.g., if we take f such that:

$$f \ : \ \forall x : \mathsf{Type}(2),\ \mathsf{Type}(1)$$

then

$$\lambda x : \mathsf{Type}(1).\ (f\ x) \ : \ \forall x : \mathsf{Type}(1),\ \mathsf{Type}(1)$$

We could not allow

$$\lambda x : \mathsf{Type}(1).\ (f\ x) \ \triangleright_\eta \ f$$

because the type of the reduced term $\forall x : \mathsf{Type}(2),\ \mathsf{Type}(1)$ would not be convertible to the type of the original term $\forall x : \mathsf{Type}(1),\ \mathsf{Type}(1)$.

Proof Irrelevance

It is legal to identify any two terms whose common type is a strict proposition $A : \mathsf{SProp}$. Terms in a strict propositions are therefore called *irrelevant*.

Convertibility

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

We say that two terms t_1 and t_2 are *$\beta\delta\iota\zeta\eta$-convertible*, or simply *convertible*, or *equivalent*, in the global environment E and local context Γ iff there exist terms u_1 and u_2 such that $E[\Gamma] \vdash t_1 \triangleright \ldots \triangleright u_1$ and $E[\Gamma] \vdash t_2 \triangleright \ldots \triangleright u_2$ and either u_1 and u_2 are identical up to irrelevant subterms, or they are convertible up to η-expansion, i.e. u_1 is $\lambda x : T.\ u_1'$ and $u_2 x$ is recursively convertible to u_1', or, symmetrically, u_2 is $\lambda x : T.\ u_2'$ and $u_1 x$ is recursively convertible to u_2'. We then write $E[\Gamma] \vdash t_1 =_{\beta\delta\iota\zeta\eta} t_2$.

Apart from this we consider two instances of polymorphic and cumulative (see Chapter *Polymorphic Universes*) inductive types (see below) convertible

$$E[\Gamma] \vdash t\ w_1 \ldots w_m =_{\beta\delta\iota\zeta\eta} t\ w_1' \ldots w_m'$$

if we have subtypings (see below) in both directions, i.e.,

$$E[\Gamma] \vdash t\ w_1 \ldots w_m \leq_{\beta\delta\iota\zeta\eta} t\ w_1' \ldots w_m'$$

and

$$E[\Gamma] \vdash t\ w_1' \ldots w_m' \leq_{\beta\delta\iota\zeta\eta} t\ w_1 \ldots w_m.$$

Furthermore, we consider

$$E[\Gamma] \vdash c\ v_1 \ldots v_m =_{\beta\delta\iota\zeta\eta} c'\ v_1' \ldots v_m'$$

convertible if

$$E[\Gamma] \vdash v_i =_{\beta\delta\iota\zeta\eta} v_i'$$

and we have that c and c' are the same constructors of different instances of the same inductive types (differing only in universe levels) such that

$$E[\Gamma] \vdash c\ v_1 \ldots v_m : t\ w_1 \ldots w_m$$

and

$$E[\Gamma] \vdash c'\ v_1' \ldots v_m' : t'\ w_1' \ldots w_m'$$

and we have

$$E[\Gamma] \vdash t\ w_1 \ldots w_m =_{\beta\delta\iota\zeta\eta} t\ w_1' \ldots w_m'.$$

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

4.4.4 Subtyping rules

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 is the *cumulativity* rule of Cic). This property extends the equivalence relation of convertibility into a *subtyping* relation inductively defined by:

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

3. for any i, $E[\Gamma] \vdash \mathsf{Set} \leq_{\beta\delta\iota\zeta\eta} \mathsf{Type}(i)$,
4. $E[\Gamma] \vdash \mathsf{Prop} \leq_{\beta\delta\iota\zeta\eta} \mathsf{Set}$, hence, by transitivity, $E[\Gamma] \vdash \mathsf{Prop} \leq_{\beta\delta\iota\zeta\eta} \mathsf{Type}(i)$, for any i (note: SProp is not related by cumulativity to any other term)
5. if $E[\Gamma] \vdash T =_{\beta\delta\iota\zeta\eta} U$ and $E[\Gamma :: (x : T)] \vdash T' \leq_{\beta\delta\iota\zeta\eta} U'$ then $E[\Gamma] \vdash \forall x : T,\ T' \leq_{\beta\delta\iota\zeta\eta} \forall x : U,\ U'$.
6. if $\mathsf{Ind}\ [p]\ (\Gamma_I\ := \ \Gamma_C)$ is a universe polymorphic and cumulative (see Chapter *Polymorphic Universes*) inductive type (see below) and $(t : \forall \Gamma_P, \forall \Gamma_{Arr(t)}, S) \in \Gamma_I$ and $(t' : \forall \Gamma'_P, \forall \Gamma'_{Arr(t)}, S') \in \Gamma_I$ are two different instances of *the same* inductive type (differing only in universe levels) with constructors

$$[c_1 : \forall \Gamma_P, \forall T_{1,1} ... T_{1,n_1},\ t\ v_{1,1} ... v_{1,m};\ ...;\ c_k : \forall \Gamma_P, \forall T_{k,1} ... T_{k,n_k},\ t\ v_{k,1} ... v_{k,m}]$$

and

$$[c_1 : \forall \Gamma'_P, \forall T'_{1,1} ... T'_{1,n_1},\ t'\ v'_{1,1} ... v'_{1,m};\ ...;\ c_k : \forall \Gamma'_P, \forall T'_{k,1} ... T'_{k,n_k},\ t'\ v'_{k,1} ... v'_{k,m}]$$

respectively then

$$E[\Gamma] \vdash t\ w_1 ... w_m \leq_{\beta\delta\iota\zeta\eta} t'\ w'_1 ... w'_m$$

(notice that t and t' are both fully applied, i.e., they have a sort as a type) if

$$E[\Gamma] \vdash w_i =_{\beta\delta\iota\zeta\eta} w'_i$$

for $1 \leq i \leq m$ and we have

$$E[\Gamma] \vdash T_{i,j} \leq_{\beta\delta\iota\zeta\eta} T'_{i,j}$$

and

$$E[\Gamma] \vdash A_i \leq_{\beta\delta\iota\zeta\eta} A'_i$$

where $\Gamma_{Arr(t)} = [a_1 : A_1;\ ...;\ a_l : A_l]$ and $\Gamma'_{Arr(t)} = [a_1 : A'_1;\ ...;\ a_l : A'_l]$.

The conversion rule up to subtyping is now exactly:

Conv

$$\frac{E[\Gamma] \vdash U : s \qquad E[\Gamma] \vdash t : T \qquad E[\Gamma] \vdash T \leq_{\beta\delta\iota\zeta\eta} U}{E[\Gamma] \vdash t : U}$$

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 rules. Among them, we have to mention the *head reduction* which will play an important role (see Chapter *Tactics*). Any term t can be written as $\lambda x_1 : T_1.\ ...\lambda x_k : T_k.\ (t_0\ t_1 ... t_n)$ 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:

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

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:

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

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.

4.4.5 Inductive Definitions

Formally, we can represent any *inductive definition* as $\mathsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C)$ where:

- Γ_I determines the names and types of inductive types;
- Γ_C determines the names and types of constructors of these inductive types;
- p determines the number of parameters of these inductive types.

These inductive definitions, together with global assumptions and global definitions, then form the global environment. Additionally, for any p there always exists $\Gamma_P = [a_1 : A_1; \; \ldots; \; a_p : A_p]$ such that each T in $(t : T) \in \Gamma_I \cup \Gamma_C$ can be written as: $\forall \Gamma_P, T'$ where Γ_P is called the *context of parameters*. Furthermore, we must have that each T in $(t : T) \in \Gamma_I$ can be written as: $\forall \Gamma_P, \forall \Gamma_{Arr(t)}, S$ where $\Gamma_{Arr(t)}$ is called the *Arity* of the inductive type t and S is called the sort of the inductive type t (not to be confused with $\mathcal{S}$ which is the set of sorts).

Example

The declaration for parameterized lists is:

$$\mathsf{Ind}\,[1]\left([\mathsf{list} : \mathsf{Set} \to \mathsf{Set}] \;:=\; \left[\begin{array}{rcl} \mathsf{nil} & : & \forall A : \mathsf{Set},\ \mathsf{list}\ A \\ \mathsf{cons} & : & \forall A : \mathsf{Set},\ A \to \mathsf{list}\ A \to \mathsf{list}\ A \end{array}\right]\right)$$

which corresponds to the result of the Coq declaration:

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

Example

The declaration for a mutual inductive definition of tree and forest is:

$$\mathsf{Ind}\,[0]\left(\left[\begin{array}{rcl} \mathsf{tree} & : & \mathsf{Set} \\ \mathsf{forest} & : & \mathsf{Set} \end{array}\right] \;:=\; \left[\begin{array}{rcl} \mathsf{node} & : & \mathsf{forest} \to \mathsf{tree} \\ \mathsf{emptyf} & : & \mathsf{forest} \\ \mathsf{consf} & : & \mathsf{tree} \to \mathsf{forest} \to \mathsf{forest} \end{array}\right]\right)$$

which corresponds to the result of the Coq declaration:

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

Example

The declaration for a mutual inductive definition of even and odd is:

$$\mathsf{Ind}\,[0]\left(\left[\begin{array}{rcl} \mathsf{even} & : & \mathsf{nat} \to \mathsf{Prop} \\ \mathsf{odd} & : & \mathsf{nat} \to \mathsf{Prop} \end{array}\right] \;:=\; \left[\begin{array}{rcl} \mathsf{even}_\mathsf{O} & : & \mathsf{even}\ 0 \\ \mathsf{even}_\mathsf{S} & : & \forall n,\ \mathsf{odd}\ n \to \mathsf{even}\ (\mathsf{S}\ n) \\ \mathsf{odd}_\mathsf{S} & : & \forall n,\ \mathsf{even}\ n \to \mathsf{odd}\ (\mathsf{S}\ n) \end{array}\right]\right)$$

which corresponds to the result of the Coq declaration:

```
Inductive even : nat -> Prop :=
| even_O : even 0
| even_S : forall n, odd n -> even (S n)
with odd : nat -> Prop :=
| odd_S : forall n, even n -> odd (S n).
```

Types of inductive objects

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

Ind

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad \mathsf{Ind}~[p]~(\Gamma_I \;:=\; \Gamma_C) \in E \qquad (a : A) \in \Gamma_I}{E[\Gamma] \vdash a : A}$$

Constr

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad \mathsf{Ind}~[p]~(\Gamma_I \;:=\; \Gamma_C) \in E \qquad (c : C) \in \Gamma_C}{E[\Gamma] \vdash c : C}$$

Example

Provided that our environment E contains inductive definitions we showed before, these two inference rules above enable us to conclude that:

$$\begin{array}{l}
E[\Gamma] \vdash \mathsf{even} : \mathsf{nat} \to \mathsf{Prop} \\
E[\Gamma] \vdash \mathsf{odd} : \mathsf{nat} \to \mathsf{Prop} \\
E[\Gamma] \vdash \mathsf{even}_\mathsf{O} : \mathsf{even}~\mathsf{O} \\
E[\Gamma] \vdash \mathsf{even}_\mathsf{S} : \forall n : \mathsf{nat},~\mathsf{odd}~n \to \mathsf{even}~(\mathsf{S}~n) \\
E[\Gamma] \vdash \mathsf{odd}_\mathsf{S} : \forall n : \mathsf{nat},~\mathsf{even}~n \to \mathsf{odd}~(\mathsf{S}~n)
\end{array}$$

Well-formed inductive definitions

We cannot accept any inductive definition 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:

Arity of a given sort

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.

Example

$A \to \mathsf{Set}$ is an arity of sort Set. $\forall A : \mathsf{Prop},~A \to \mathsf{Prop}$ is an arity of sort Prop.

Arity

A type T is an *arity* if there is a $s \in \mathcal{S}$ such that T is an arity of sort s.

Example

$A \rightarrow \mathsf{Set}$ and $\forall A : \mathsf{Prop},\ A \rightarrow \mathsf{Prop}$ are arities.

Type of constructor

We say that T is a *type of constructor of* I in one of the following two cases:

- T is $(I\ t_1 \ldots t_n)$
- T is $\forall x : U,\ T'$ where T' is also a type of constructor of I

Example

nat and $\mathsf{nat} \rightarrow \mathsf{nat}$ are types of constructor of nat. $\forall A : \mathsf{Type},\ \mathsf{list}\ A$ and $\forall A : \mathsf{Type},\ A \rightarrow \mathsf{list}\ A \rightarrow \mathsf{list}\ A$ are types of constructor of list.

Positivity Condition

The type of constructor T will be said to *satisfy the positivity condition* for a constant X in the following cases:

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

Strict positivity

The constant X *occurs strictly positively* in T in the following cases:

- X does not occur in T
- T converts to $(X\ t_1 \ldots t_n)$ and X does not occur in any of t_i
- T converts to $\forall x : U,\ V$ and X does not occur in type U but occurs strictly positively in type V
- T converts to $(I\ a_1 \ldots a_m\ t_1 \ldots t_p)$ where I is the name of an inductive definition of the form

  $$\mathsf{Ind}\ [m]\,(I : A\ :=\ c_1 : \forall p_1 : P_1, \ldots \forall p_m : P_m,\ C_1;\ \ldots;\ c_n : \forall p_1 : P_1, \ldots \forall p_m : P_m,\ C_n)$$

  (in particular, it is not mutually defined and it has m parameters) and X does not occur in any of the t_i, and the (instantiated) types of constructor $C_i\{p_j/a_j\}_{j=1\ldots m}$ of I satisfy the nested positivity condition for X

Nested Positivity

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

- $T = (I\ b_1 \ldots b_m\ u_1 \ldots u_p)$, I is an inductive type with m parameters and X does not occur in any u_i
- $T = \forall x : U,\ V$ and X occurs only strictly positively in U and the type V satisfies the nested positivity condition for X

Example

For instance, if one considers the following variant of a tree type branching over the natural numbers:

```
Inductive nattree (A:Type) : Type :=
| leaf : nattree A
| natnode : A -> (nat -> nattree A) -> nattree A.
```

Then every instantiated constructor of `nattree A` satisfies the nested positivity condition for `nattree`:

- Type `nattree A` of constructor `leaf` satisfies the positivity condition for `nattree` because `nattree` does not appear in any (real) arguments of the type of that constructor (primarily because `nattree` does not have any (real) arguments) ... (bullet 1)
- Type `A` → `(nat` → `nattree A)` → `nattree A` of constructor `natnode` satisfies the positivity condition for `nattree` because:
  - `nattree` occurs only strictly positively in `A` ... (bullet 1)
  - `nattree` occurs only strictly positively in `nat` → `nattree A` ... (bullet 3 + 2)
  - `nattree` satisfies the positivity condition for `nattree A` ... (bullet 1)

Correctness rules

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

Let E be a global environment and Γ_P, Γ_I, Γ_C be contexts such that Γ_I is $[I_1 : \forall \Gamma_P, A_1;\ \ldots;\ I_k : \forall \Gamma_P, A_k]$, and Γ_C is $[c_1 : \forall \Gamma_P, C_1;\ \ldots;\ c_n : \forall \Gamma_P, C_n]$. Then

W-Ind

$$\frac{\mathcal{WF}(E)[\Gamma_P] \qquad (E[\Gamma_I; \Gamma_P] \vdash C_i : s_{q_i})_{i=1\ldots n}}{\mathcal{WF}(E;\ \mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C))[]}$$

provided that the following side conditions hold:

- $k > 0$ and all of I_j and c_i are distinct names for $j = 1 \ldots k$ and $i = 1 \ldots n$,
- p is the number of parameters of $\mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C)$ and Γ_P is the context of parameters,
- for $j = 1 \ldots k$ we have that A_j is an arity of sort s_j and $I_j \notin E$,
- for $i = 1 \ldots n$ we have that C_i is a type of constructor of I_{q_i} which satisfies the positivity condition for $I_1 \ldots I_k$ and $c_i \notin E$.

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 sorts SProp and Prop but may fail to define inductive type on sort Set and generate constraints between universes for inductive types in the Type hierarchy.

Example

It is well known that the existential quantifier can be encoded as an inductive definition. The following declaration introduces the second-order existential quantifier $\exists X.P(X)$.

```
Inductive exProp (P:Prop->Prop) : Prop :=
| exP_intro : forall X:Prop, P X -> exProp P.
```

The same definition on Set is not allowed and fails:

```
Fail Inductive exSet (P:Set->Prop) : Set :=
exS_intro : forall X:Set, P X -> exSet P.
    The command has indeed failed with message:
    Large non-propositional inductive types must be in Type.
```

It is possible to declare the same inductive definition in the universe Type. The `exType` inductive definition has type $(\mathsf{Type}(i) \to \mathsf{Prop}) \to \mathsf{Type}(j)$ with the constraint that the parameter X of $\mathsf{exT}_{\mathsf{intro}}$ has type $\mathsf{Type}(k)$ with $k < j$ and $k \leq i$.

```
Inductive exType (P:Type->Prop) : Type :=
exT_intro : forall X:Type, P X -> exType P.
    exType is defined
    exType_rect is defined
    exType_ind is defined
    exType_rec is defined
    exType_sind is defined
```

Example: Negative occurrence (first example)

The following inductive definition is rejected because it does not satisfy the positivity condition:

```
Fail Inductive I : Prop := not_I_I (not_I : I -> False) : I.
    The command has indeed failed with message:
    Non strictly positive occurrence of "I" in "(I -> False) -> I".
```

If we were to accept such definition, we could derive a contradiction from it (we can test this by disabling the *Positivity Checking* flag):

```
Definition I_not_I : I -> ~ I := fun i =>
  match i with not_I_I not_I => not_I end.
    I_not_I is defined
```

```
Lemma contradiction : False.
Proof.
enough (I /\ ~ I) as [] by contradiction.
split.
- apply not_I_I.
intro.
now apply I_not_I.
- intro.
now apply I_not_I.
Qed.
```

Example: Negative occurrence (second example)

Here is another example of an inductive definition which is rejected because it does not satify the positivity condition:

```
Fail Inductive Lam := lam (_ : Lam -> Lam).
    The command has indeed failed with message:
    Non strictly positive occurrence of "Lam" in "(Lam -> Lam) -> Lam".
```

Again, if we were to accept it, we could derive a contradiction (this time through a non-terminating recursive function):

```
Fixpoint infinite_loop l : False :=
  match l with lam x => infinite_loop (x l) end.
    infinite_loop is defined
    infinite_loop is recursively defined (decreasing on 1st argument)

Check infinite_loop (lam (@id Lam)) : False.
    infinite_loop (lam (id (A:=Lam))) : False
         : False
```

Example: Non strictly positive occurrence

It is less obvious why inductive type definitions with occurences that are positive but not strictly positive are harmful. We will see that in presence of an impredicative type they are unsound:

```
Fail Inductive A: Type := introA: ((A -> Prop) -> Prop) -> A.
    The command has indeed failed with message:
    Non strictly positive occurrence of "A" in "((A -> Prop) -> Prop) -> A".
```

If we were to accept this definition we could derive a contradiction by creating an injective function from $A \rightarrow \mathsf{Prop}$ to A.

This function is defined by composing the injective constructor of the type A with the function $\lambda x.\lambda z.z = x$ injecting any type T into $T \rightarrow \mathsf{Prop}$.

```
Definition f (x: A -> Prop): A := introA (fun z => z = x).
    f is defined


Lemma f_inj: forall x y, f x = f y -> x = y.
Proof.
unfold f; intros ? ? H; injection H.
set (F := fun z => z = y); intro HF.
symmetry; replace (y = x) with (F y).
+ unfold F; reflexivity.
+ rewrite <- HF; reflexivity.
Qed.
```

The type $A \rightarrow \mathsf{Prop}$ can be understood as the powerset of the type A. To derive a contradiction from the injective function f we use Cantor's classic diagonal argument.

```
Definition d: A -> Prop := fun x => exists s, x = f s /\ ~s x.
    d is defined

Definition fd: A := f d.
    fd is defined
```

```
Lemma cantor: (d fd) <-> ~(d fd).
Proof.
split.
+ intros [s [H1 H2]]; unfold fd in H1.
replace d with s.
* assumption.
* apply f_inj; congruence.
+ intro; exists d; tauto.
Qed.
Lemma bad: False.
Proof.
pose cantor; tauto.
Qed.
```

This derivation was first presented by Thierry Coquand and Christine Paulin in *[CP90]*.

Template polymorphism

Inductive types can be made polymorphic over the universes introduced by their parameters in Type, if the minimal inferred sort of the inductive declarations either mention some of those parameter universes or is computed to be Prop or Set.

If A is an arity of some sort and s is a sort, we write $A_{/s}$ for the arity obtained from A by replacing its sort with s. Especially, if A is well-typed in some global environment and local context, then $A_{/s}$ is typable by typability of all products in the Calculus of Inductive Constructions. The following typing rule is added to the theory.

Let $\mathsf{Ind}\,[p]\,(\Gamma_I \,:=\, \Gamma_C)$ be an inductive definition. Let $\Gamma_P = [p_1 : P_1;\ \ldots;\ p_p : P_p]$ be its context of parameters, $\Gamma_I = [I_1 : \forall\Gamma_P, A_1;\ \ldots;\ I_k : \forall\Gamma_P, A_k]$ its context of definitions and $\Gamma_C = [c_1 : \forall\Gamma_P, C_1;\ \ldots;\ c_n : \forall\Gamma_P, C_n]$ its context of constructors, with c_i a constructor of I_{q_i}. Let $m \leq p$ be the length of the longest prefix of parameters such that the m first arguments of all occurrences of all I_j in all C_k (even the occurrences in the hypotheses of C_k) are exactly applied to $p_1 \ldots p_m$ (m is the number of *recursively uniform parameters* and the $p - m$ remaining parameters are the *recursively non-uniform parameters*). Let $q_1, \ldots, q_r$, with $0 \leq r \leq m$, be a (possibly) partial instantiation of the recursively uniform parameters of Γ_P. We have:

Ind-Family

$$\frac{\left\{\begin{array}{l} \mathsf{Ind}\,[p]\,(\Gamma_I \,:=\, \Gamma_C) \in E \\ (E[] \vdash q_l : P'_l)_{l=1\ldots r} \\ (E[] \vdash P'_l \leq_{\beta\delta\iota\zeta\eta} P_l\{p_u/q_u\}_{u=1\ldots l-1})_{l=1\ldots r} \\ 1 \leq j \leq k \end{array}\right.}{E[] \vdash I_j\ q_1 \ldots q_r : \forall[p_{r+1} : P_{r+1};\ \ldots;\ p_p : P_p], (A_j)_{/s_j}}$$

provided that the following side conditions hold:

- $\Gamma_{P'}$ is the context obtained from Γ_P by replacing each P_l that is an arity with P'_l for $1 \leq l \leq r$ (notice that P_l arity implies P'_l arity since $E[] \vdash P'_l \leq_{\beta\delta\iota\zeta\eta} P_l\{p_u/q_u\}_{u=1\ldots l-1}$);
- there are sorts s_i, for $1 \leq i \leq k$ such that, for $\Gamma_{I'} = [I_1 : \forall\Gamma_{P'}, (A_1)_{/s_1};\ \ldots;\ I_k : \forall\Gamma_{P'}, (A_k)_{/s_k}]$ we have $(E[\Gamma_{I'}; \Gamma_{P'}] \vdash C_i : s_{q_i})_{i=1\ldots n}$;
- the sorts s_i are all introduced by the inductive declaration and have no universe constraints beside being greater than or equal to Prop, and such that all eliminations, to Prop, Set and Type(j), are allowed (see Section *Destructors*).

Notice that if $I_j\ q_1 \ldots q_r$ is typable using the rules **Ind-Const** and **App**, then it is typable using the rule **Ind-Family**. Conversely, the extended theory is not stronger than the theory without **Ind-Family**. We get an equiconsistency result by mapping each $\mathsf{Ind}\ [p]\ (\Gamma_I\ :=\ \Gamma_C)$ occurring into a given derivation into as many different inductive types and constructors as the number of different (partial) replacements of sorts, needed for this derivation, in the parameters that are arities (this is possible because $\mathsf{Ind}\ [p]\ (\Gamma_I\ :=\ \Gamma_C)$ well-formed implies that $\mathsf{Ind}\ [p]\ (\Gamma_{I'}\ :=\ \Gamma_{C'})$ is well-formed and has the same allowed eliminations, where $\Gamma_{I'}$ is defined as above and $\Gamma_{C'} = [c_1 : \forall\Gamma_{P'}, C_1;\ \ldots;\ c_n : \forall\Gamma_{P'}, C_n])$. That is, the changes in the types of each partial instance $q_1 \ldots q_r$ can be characterized by the ordered sets of arity sorts among the types of parameters, and to each signature is associated a new inductive definition with fresh names. Conversion is preserved as any (partial) instance $I_j\ q_1 \ldots q_r$ or $C_i\ q_1 \ldots q_r$ is mapped to the names chosen in the specific instance of $\mathsf{Ind}\ [p]\ (\Gamma_I\ :=\ \Gamma_C)$.

Warning: The restriction that sorts are introduced by the inductive declaration prevents inductive types declared in sections to be template-polymorphic on universes introduced previously in the section: they cannot parameterize over the universes introduced with section variables that become parameters at section closing time, as these may be shared with other definitions from the same section which can impose constraints on them.

Flag: `Auto Template Polymorphism`

This flag, enabled by default, makes every inductive type declared at level `Type` (without annotations or hiding it behind a definition) template polymorphic if possible.

This can be prevented using the `universes(notemplate)` attribute.

Warning: **`Automatically declaring`** ***`ident`*** **`as template polymorphic.`**

Warning `auto-template` can be used to find which types are implicitly declared template polymorphic by *`Auto Template Polymorphism`*.

An inductive type can be forced to be template polymorphic using the `universes(template)` attribute: it should then fulfill the criterion to be template polymorphic or an error is raised.

Error: **`Inductive`** ***`ident`*** **`cannot be made template polymorphic.`**

This error is raised when the `#[universes(template)]` attribute is on but the inductive cannot be made polymorphic on any universe or be inferred to live in `Prop` or `Set`.

Template polymorphism and universe polymorphism (see Chapter *Polymorphic Universes*) are incompatible, so if the later is enabled it will prevail over automatic template polymorphism and cause an error when using the `universes(template)` attribute.

Flag: `Template Check`

This flag is on by default. Turning it off disables the check of locality of the sorts when abstracting the inductive over its parameters. This is a deprecated and *unsafe* flag that can introduce inconsistencies, it is only meant to help users incrementally update code from Coq versions < 8.10 which did not implement this check. The `Coq89.v` compatibility file sets this flag globally. A global `-no-template-check` command line option is also available. Use at your own risk. Use of this flag is recorded in the typing flags associated to a definition but is *not* supported by the Coq checker (`coqchk`). It will appear in `Print Assumptions` and `About @ident` output involving inductive declarations that were (potentially unsoundly) assumed to be template polymorphic.

In practice, the rule **Ind-Family** is used by Coq only when all the inductive types of the inductive definition are declared with an arity whose sort is in the Type hierarchy. Then, the polymorphism is over the parameters whose type is an arity of sort in the Type hierarchy. The sorts s_j are chosen canonically so that each s_j is minimal with respect to the hierarchy $\mathsf{Prop} \subset \mathsf{Set}_p \subset \mathsf{Type}$ where Set_p is predicative Set. More precisely, an empty or small singleton inductive definition (i.e. an inductive definition of which all inductive types are singleton – see Section *Destructors*) is set in Prop, a small non-singleton inductive type is set in Set (even in case Set is impredicative – see Section *The-Calculus-of-Inductive-Construction-with-impredicative-Set*), and otherwise in the Type hierarchy.

Note that the side-condition about allowed elimination sorts in the rule **Ind-Family** avoids to recompute the allowed elimination sorts at each instance of a pattern matching (see Section *Destructors*). As an example, let us consider the following definition:

Example

```
Inductive option (A:Type) : Type :=
| None : option A
| Some : A -> option A.
```

As the definition is set in the Type hierarchy, it is used polymorphically over its parameters whose types are arities of a sort in the Type hierarchy. Here, the parameter A has this property, hence, if `option` is applied to a type in Set, the result is in Set. Note that if `option` is applied to a type in Prop, then, the result is not set in Prop but in Set still. This is because `option` is not a singleton type (see Section *Destructors*) and it would lose the elimination to Set and Type if set in Prop.

Example

```
Check (fun A:Set => option A).
    fun A : Set => option A
         : Set -> Set

Check (fun A:Prop => option A).
    fun A : Prop => option A
         : Prop -> Set
```

Here is another example.

Example

```
Inductive prod (A B:Type) : Type := pair : A -> B -> prod A B.
```

As `prod` is a singleton type, it will be in Prop if applied twice to propositions, in Set if applied twice to at least one type in Set and none in Type, and in Type otherwise. In all cases, the three kind of eliminations schemes are allowed.

Example

```
Check (fun A:Set => prod A).
    fun A : Set => prod A
         : Set -> Type -> Type

Check (fun A:Prop => prod A A).
    fun A : Prop => prod A A
         : Prop -> Prop

Check (fun (A:Prop) (B:Set) => prod A B).
    fun (A : Prop) (B : Set) => prod A B
```

(continues on next page)

(continued from previous page)

```
        : Prop -> Set -> Set

Check (fun (A:Type) (B:Prop) => prod A B).
    fun (A : Type) (B : Prop) => prod A B
        : Type -> Prop -> Type
```

Note: Template polymorphism used to be called "sort-polymorphism of inductive types" before universe polymorphism (see Chapter *Polymorphic Universes*) was introduced.

Destructors

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

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

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

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

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

From the logical point of view, we have built a type family by giving a set of constructors. We want to capture the fact that we do not have any other way to build an object in this type. So when trying to prove a property about an object m in an inductive type 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 $\forall l : \mathsf{list}\ A,\ (\mathsf{has\_length}\ A\ l\ (\mathsf{length}\ l))$ it is enough to prove:

- $(\mathsf{has\_length}\ A\ (\mathsf{nil}\ A)\ (\mathsf{length}\ (\mathsf{nil}\ A)))$
- $\forall a : A,\ \forall l : \mathsf{list}\ A,\ (\mathsf{has\_length}\ A\ l\ (\mathsf{length}\ l)) \to (\mathsf{has\_length}\ A\ (\mathsf{cons}\ A\ a\ l)\ (\mathsf{length}\ (\mathsf{cons}\ A\ a\ l)))$

which given the conversion equalities satisfied by length is the same as proving:

- $(\mathsf{has\_length}\ A\ (\mathsf{nil}\ A)\ \mathsf{O})$
- $\forall a : A,\ \forall l : \mathsf{list}\ A,\ (\mathsf{has\_length}\ A\ l\ (\mathsf{length}\ l)) \to (\mathsf{has\_length}\ A\ (\mathsf{cons}\ A\ a\ l)\ (\mathsf{S}\ (\mathsf{length}\ l)))$

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

But this operator is rather tedious to implement and use. We choose in this version of Coq to factorize the operator for primitive recursion into two more primitive operations as was first suggested by Th. Coquand in *[Coq92]*. 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 operator is that we have an object m in an inductive type I and we want to prove a property which possibly depends on m. For this, it is enough to prove the property for $m = (c_i\ u_1 \ldots u_{p_i})$ for each constructor of I. The Coq term for this proof will be written:

$$\mathsf{match}\ m\ \mathsf{with}\ (c_1\ x_{11} \ldots x_{1p_1}) \Rightarrow f_1 | \ldots | (c_n\ x_{n1} \ldots x_{np_n}) \Rightarrow f_n\ \mathsf{end}$$

In this expression, if m eventually happens to evaluate to $(c_i\ u_1 \ldots u_{p_i})$ then the expression will behave as specified in its i-th branch and it will reduce to f_i where the $x_{i1} \ldots x_{ip_i}$ are replaced by the $u_1 \ldots u_{p_i}$ according to the ι-reduction.

Actually, for type checking a match…with…end expression we also need to know the predicate P to be proved by case analysis. In the general case where I is an inductively defined n-ary relation, P is a predicate over $n+1$ arguments: the n first ones correspond to the arguments of I (parameters excluded), and the last one corresponds to object m. Coq can sometimes infer this predicate but sometimes not. The concrete syntax for describing this predicate uses the as…in…return construction. For instance, let us assume that I is an unary predicate with one parameter and one argument. The predicate is made explicit using the syntax:

$$\mathsf{match}\ m\ \mathsf{as}\ x\ \mathsf{in}\ I\ \_\ a\ \mathsf{return}\ P\ \mathsf{with}\ (c_1\ x_{11} \ldots x_{1p_1}) \Rightarrow f_1 | \ldots | (c_n\ x_{n1} \ldots x_{np_n}) \Rightarrow f_n\ \mathsf{end}$$

The as part can be omitted if either the result type does not depend on m (non-dependent elimination) or m is a variable (in this case, m can occur in P where it is considered a bound variable). The in part can be omitted if the result type does not depend on the arguments of I. Note that the arguments of I corresponding to parameters *must* be _, because the result type is not generalized to all possible values of the parameters. The other arguments of I (sometimes called indices in the literature) have to be variables (a above) and these variables can occur in P. The expression after in must be seen as an *inductive type pattern*. Notice that expansion of implicit arguments and notations apply to this pattern. For the purpose of presenting the inference rules, we use a more compact notation:

$$\mathsf{case}(m, (\lambda a x.P), \lambda x_{11} \ldots x_{1p_1}.f_1\ | \ldots |\ \lambda x_{n1} \ldots x_{np_n}.f_n)$$

Allowed elimination sorts. An important question for building the typing rule for match is what can be the type of $\lambda a x.P$ with respect to the type of m. If $m : I$ and $I : A$ and $\lambda a x.P : B$ then by $[I : A|B]$ we mean that one can use $\lambda a x.P$ with m in the above match-construct.

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

The case of inductive types in sorts Set or Type is simple. There is no restriction on the sort of the predicate to be eliminated.

Prod

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

Set & Type

$$\frac{s_1 \in \{\mathsf{Set}, \mathsf{Type}(j)\} \qquad s_2 \in \mathcal{S}}{[I : s_1|I \to s_2]}$$

The case of Inductive definitions of sort Prop is a bit more complicated, because of our interpretation of this sort. The only harmless allowed eliminations, are the ones when predicate P is also of sort Prop or is of the morally smaller sort SProp.

Prop

$$\frac{s \in \{\mathsf{SProp}, \mathsf{Prop}\}}{[I : \mathsf{Prop} | I \to s]}$$

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:

Example

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

The following definition which computes a boolean value by case over the proof of `or A B` is not accepted:

Example

```
Fail Definition choice (A B: Prop) (x:or A B) :=
match x with or_introl _ _ a => true | or_intror _ _ b => false end.
    The command has indeed failed with message:
    Incorrect elimination of "x" in the inductive type "or":
    the return type has sort "Set" while it should be "SProp" or "Prop".
    Elimination of an inductive object of sort Prop
    is not allowed on a predicate in sort Set
    because proofs can be eliminated only to build proofs.
```

From the computational point of view, the structure of the proof of `(or A B)` in this term is needed for computing the boolean value.

In general, if I has type Prop then P cannot have type $I \to$ Set, because it will mean to build an informative proof of type $(P\ m)$ doing a case analysis over a non-computational object that will disappear in the extracted program. But the other way is safe with respect to our interpretation we can have I a computational object and P a non-computational one, it just corresponds to proving a logical property of a computational object.

In the same spirit, elimination on P of type $I \to$ Type cannot be allowed because it trivially implies the elimination on P of type $I \to$ Set by cumulativity. It also implies that there are two proofs of the same property which are provably different, contradicting the proof-irrelevance property which is sometimes a useful axiom:

Example

```
Axiom proof_irrelevance : forall (P : Prop) (x y : P), x=y.
    proof_irrelevance is declared
```

The elimination of an inductive type of sort Prop on a predicate P of type $I \to$ Type leads to a paradox when applied to impredicative inductive definition like the second-order existential quantifier `exProp` defined above, because it gives 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

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

A *singleton definition* has only one constructor and all the arguments of this constructor have type Prop. In that case, there is a canonical way to interpret the informative extraction on an object in that type, such that the elimination on any sort s is legal. Typical examples are the conjunction of non-informative propositions and the equality. If there is a hypothesis $h : a = b$ in the local context, it can be used for rewriting not only in logical propositions but also in any type.

Example

```
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

    Arguments eq_rec [A]%type_scope _ _%function_scope

Require Extraction.
    [Loading ML file extraction_plugin.cmxs ... done]

Extraction eq_rec.
    (** val eq_rec : 'a1 -> 'a2 -> 'a1 -> 'a2 **)

    let eq_rec _ f _ =
      f
```

An empty definition has no constructors, in that case also, elimination on any sort is allowed.

Inductive types in SProp must have no constructors (i.e. be empty) to be eliminated to produce relevant values.

Note that thanks to proof irrelevance elimination functions can be produced for other types, for instance the elimination for a unit type is the identity.

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

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

$$\begin{array}{ll} \{c : (I\ q_1 \ldots q_r\ t_1 \ldots t_s)\}^P & \equiv (P\ t_1 \ldots\ t_s\ c) \\ \{c : \forall x : T,\ C\}^P & \equiv \forall x : T,\ \{(c\ x) : C\}^P \end{array}$$

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

Example

The following term in concrete syntax:

```
match t as l return P' with
| nil _ => t1
| cons _ hd tl => t2
end
```

can be represented in abstract syntax as

$$\mathsf{case}(t, P, f_1 | f_2)$$

where

$$\begin{array}{rcl} P & = & \lambda l.\ P' \\ f_1 & = & t_1 \\ f_2 & = & \lambda(hd : \mathsf{nat}).\ \lambda(tl : \mathsf{list\ nat}).\ t_2 \end{array}$$

According to the definition:

$$\{(\mathsf{nil\ nat})\}^P \equiv \{(\mathsf{nil\ nat}) : (\mathsf{list\ nat})\}^P \equiv (P\ (\mathsf{nil\ nat}))$$

$$\begin{array}{rl} \{(\mathsf{cons\ nat})\}^P & \equiv \{(\mathsf{cons\ nat}) : (\mathsf{nat} \to \mathsf{list\ nat} \to \mathsf{list\ nat})\}^P \\ & \equiv \forall n : \mathsf{nat},\ \{(\mathsf{cons\ nat}\ n) : (\mathsf{list\ nat} \to \mathsf{list\ nat})\}^P \\ & \equiv \forall n : \mathsf{nat},\ \forall l : \mathsf{list\ nat},\ \{(\mathsf{cons\ nat}\ n\ l) : (\mathsf{list\ nat})\}^P \\ & \equiv \forall n : \mathsf{nat},\ \forall l : \mathsf{list\ nat},\ (P\ (\mathsf{cons\ nat}\ n\ l)). \end{array}$$

Given some P then $\{(\mathsf{nil\ nat})\}^P$ represents the expected type of f_1, and $\{(\mathsf{cons\ nat})\}^P$ represents the expected type of f_2.

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

match

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

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

Example

Below is a typing rule for the term shown in the previous example:

list example

$$\frac{\begin{array}{l} E[\Gamma] \vdash t : (\mathsf{list\ nat}) \\ E[\Gamma] \vdash P : B \\ [(\mathsf{list\ nat}) | B] \\ E[\Gamma] \vdash f_1 : \{(\mathsf{nil\ nat})\}^P \\ E[\Gamma] \vdash f_2 : \{(\mathsf{cons\ nat})\}^P \end{array}}{E[\Gamma] \vdash \mathsf{case}(t, P, f_1 | f_2) : (P\ t)}$$

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

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

$$\mathsf{case}((c_{p_i}\ q_1 \ldots q_r\ a_1 \ldots a_m), P, f_1|\ldots|f_l)$$

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

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

$$\mathsf{case}((c_{p_i}\ q_1 \ldots q_r\ a_1 \ldots a_m), P, f_1|\ldots|f_l) \triangleright_\iota (f_i\ a_1 \ldots a_m)$$

Fixpoint definitions

The second operator for elimination is fixpoint definition. This fixpoint may involve several mutually recursive definitions. The basic concrete syntax for a recursive set of mutually recursive declarations is (with Γ_i contexts):

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

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

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

In the inference rules, we represent such a term by

$$\mathsf{Fix}\ f_i\{f_1 : A'_1 := t'_1 \ldots f_n : A'_n := t'_n\}$$

with t'_i (resp. A'_i) representing the term t_i abstracted (resp. generalized) with respect to the bindings in the context Γ_i, namely $t'_i = \lambda\Gamma_i.t_i$ and $A'_i = \forall\Gamma_i, A_i$.

Typing rule

The typing rule is the expected one for a fixpoint.

Fix

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

Any fixpoint definition cannot be accepted because non-normalizing terms allow proofs of absurdity. The basic scheme of recursion that should be allowed is the one needed for defining primitive recursive functionals. In that case the fixpoint enjoys a special syntactic restriction, namely one of the arguments belongs to an inductive type, the function starts with a case analysis and recursive calls are done on variables coming from patterns and representing subterms. For instance in the case of natural numbers, a proof of the induction principle of type

$$\forall P : \mathsf{nat} \to \mathsf{Prop},\ (P\ \mathsf{O}) \to (\forall n : \mathsf{nat},\ (P\ n) \to (P\ (\mathsf{S}\ n))) \to \forall n : \mathsf{nat},\ (P\ n)$$

can be represented by the term:

$$\begin{array}{l}\lambda P : \mathsf{nat} \to \mathsf{Prop}.\ \lambda f : (P\ \mathsf{O}).\ \lambda g : (\forall n : \mathsf{nat},\ (P\ n) \to (P\ (\mathsf{S}\ n))).\\ \mathsf{Fix}\ h\{h : \forall n : \mathsf{nat},\ (P\ n) := \lambda n : \mathsf{nat}.\ \mathsf{case}(n, P, f|\lambda p : \mathsf{nat}.\ (g\ p\ (h\ p)))\}\end{array}$$

Before accepting a fixpoint definition as being correctly typed, we check that the definition is "guarded". A precise analysis of this notion can be found in *[Gimenez94]*. The first stage is to precise on which argument the fixpoint will be decreasing. The type of this argument should be an inductive type. For doing this, the syntax of fixpoints is extended and becomes

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

where k_i are positive integers. Each k_i represents the index of parameter of f_i, on which f_i is decreasing. Each A_i should be a type (reducible to a term) starting with at least k_i products $\forall y_1 : B_1,\ \ldots\forall y_{k_i} : B_{k_i},\ A'_i$ and B_{k_i} an inductive type.

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

The definition of being structurally smaller is a bit technical. One needs first to define the notion of *recursive arguments of a constructor*. For an inductive definition $\mathsf{Ind}\,[r]\,(\Gamma_I := \Gamma_C)$, if the type of a constructor c has the form $\forall p_1 : P_1,\ \ldots\forall p_r : P_r,\ \forall x_1 : T_1,\ \ldots\forall x_m : T_m,\ (I_j\ p_1 \ldots p_r\ t_1 \ldots t_s)$, then 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 an inductively defined type in a declaration $\mathsf{Ind}\,[r]\,(\Gamma_I := \Gamma_C)$ where Γ_I is $[I_1 : A_1;\ \ldots;\ I_k : A_k]$, and Γ_C is $[c_1 : C_1;\ \ldots;\ c_n : C_n]$, the terms structurally smaller than y are:

- $(t\ u)$ and $\lambda x : U.\ t$ when t is structurally smaller than y.
- $\mathsf{case}(c, P, f_1 \ldots f_n)$ when each f_i is structurally smaller than y. If c is y or is structurally smaller than y, its type is an inductive type I_p part of the inductive definition corresponding to y. Each f_i corresponds to a type of constructor $C_q \equiv \forall p_1 : P_1,\ \ldots, \forall p_r : P_r,\ \forall y_1 : B_1,\ \ldots\forall y_m : B_m,\ (I_p\ p_1 \ldots p_r\ t_1 \ldots t_s)$ and can consequently be written $\lambda y_1 : B'_1.\ \ldots\lambda y_m : B'_m.\ g_i$. ($B'_i$ is obtained from B_i by substituting parameters for 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 and show the internal representation.

Example

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

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

    Arguments plus (_ _)%nat_scope

Fixpoint lgth (A:Set) (l:list A) {struct l} : nat :=
match l with
| nil _ => O
| cons _ a l' => S (lgth A l')
```

(continues on next page)

(continued from previous page)

```
end.
    lgth is defined
    lgth is recursively defined (decreasing on 2nd argument)

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

    Arguments lgth _%type_scope

Fixpoint sizet (t:tree) : nat := let (f) := t in S (sizef f)
with sizef (f:forest) : nat :=
match f with
| emptyf => O
| consf t f => plus (sizet t) (sizef f)
end.
    sizet is defined
    sizef is defined
    sizet, sizef are recursively defined (decreasing respectively on 1st,
    1st arguments)

Print sizet.
    sizet =
    fix sizet (t : tree) : nat := let (f) := t in S (sizef f)
    with sizef (f : forest) : nat :=
      match f with
      | emptyf => 0
      | consf t f0 => plus (sizet t) (sizef f0)
      end
    for sizet
         : tree -> nat
```

Reduction rule

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

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

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

$$\begin{array}{rcl} \mathsf{plus}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{O}))\ (\mathsf{S}\ \mathsf{O}) & \triangleright_\iota & \mathsf{S}\ (\mathsf{plus}\ (\mathsf{S}\ \mathsf{O})\ (\mathsf{S}\ \mathsf{O})) \\ & \triangleright_\iota & \mathsf{S}\ (\mathsf{S}\ (\mathsf{plus}\ \mathsf{O}\ (\mathsf{S}\ \mathsf{O}))) \\ & \triangleright_\iota & \mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{O})) \end{array}$$

Mutual induction

The principles of mutual induction can be automatically generated using the Scheme command described in Section *Generation of induction principles with Scheme*.

4.4.6 Admissible rules for global environments

From the original rules of the type system, one can show the admissibility of rules which change the local context of definition of objects in the global environment. We show here the admissible rules that are used in the discharge mechanism at the end of a section.

Abstraction. One can modify a global declaration by generalizing it over a previously assumed constant c. For doing that, we need to modify the reference to the global declaration in the subsequent global environment and local context by explicitly applying this constant to the constant c.

Below, if Γ is a context of the form $[y_1 : A_1; \ldots; y_n : A_n]$, we write $\forall x : U,\ \Gamma\{c/x\}$ to mean $[y_1 : \forall x : U,\ A_1\{c/x\}; \ldots; y_n : \forall x : U,\ A_n\{c/x\}]$ and $E\{|\Gamma|/|\Gamma|c\}$ to mean the parallel substitution $E\{y_1/(y_1\ c)\}\ldots\{y_n/(y_n\ c)\}$.

First abstracting property:

$$\frac{\mathcal{WF}(E;\ c : U;\ E';\ c' := t : T;\ E'')[\Gamma]}{\mathcal{WF}(E;\ c : U;\ E';\ c' := \lambda x : U.\ t\{c/x\} : \forall x : U,\ T\{c/x\};\ E''\{c'/(c'\ c)\})[\Gamma\{c'/(c'\ c)\}]}$$

$$\frac{\mathcal{WF}(E;\ c : U;\ E';\ c' : T;\ E'')[\Gamma]}{\mathcal{WF}(E;\ c : U;\ E';\ c' : \forall x : U,\ T\{c/x\};\ E''\{c'/(c'\ c)\})[\Gamma\{c'/(c'\ c)\}]}$$

$$\frac{\mathcal{WF}(E;\ c : U;\ E';\ \mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C)\,;\ E'')[\Gamma]}{\mathcal{WF}\begin{array}{l}(E;\ c : U;\ E';\ \mathsf{Ind}\ [p+1]\,(\forall x : U,\ \Gamma_I\{c/x\}\ :=\ \forall x : U,\ \Gamma_C\{c/x\})\,;\ E''\{|\Gamma_I;\Gamma_C|/|\Gamma_I;\Gamma_C|c\})\\ [\Gamma\{|\Gamma_I;\Gamma_C|/|\Gamma_I;\Gamma_C|c\}]\end{array}}$$

One can similarly modify a global declaration by generalizing it over a previously defined constant c. Below, if Γ is a context of the form $[y_1 : A_1; \ldots; y_n : A_n]$, we write $\Gamma\{c/u\}$ to mean $[y_1 : A_1\{c/u\}; \ldots; y_n : A_n\{c/u\}]$.

Second abstracting property:

$$\frac{\mathcal{WF}(E;\ c := u : U;\ E';\ c' := t : T;\ E'')[\Gamma]}{\mathcal{WF}(E;\ c := u : U;\ E';\ c' := (\mathsf{let}\ x := u : U\ \mathsf{in}\ t\{c/x\}) : T\{c/u\};\ E'')[\Gamma]}$$

$$\frac{\mathcal{WF}(E;\ c := u : U;\ E';\ c' : T;\ E'')[\Gamma]}{\mathcal{WF}(E;\ c := u : U;\ E';\ c' : T\{c/u\};\ E'')[\Gamma]}$$

$$\frac{\mathcal{WF}(E;\ c := u : U;\ E';\ \mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C)\,;\ E'')[\Gamma]}{\mathcal{WF}(E;\ c := u : U;\ E';\ \mathsf{Ind}\ [p]\,(\Gamma_I\{c/u\}\ :=\ \Gamma_C\{c/u\})\,;\ E'')[\Gamma]}$$

Pruning the local context. If one abstracts or substitutes constants with the above rules then it may happen that some declared or defined constant does not occur any more in the subsequent global environment and in the local context. One can consequently derive the following property.

First pruning property:

$$\frac{\mathcal{WF}(E;\ c : U;\ E')[\Gamma] \qquad c\ \text{does not occur in}\ E'\ \text{and}\ \Gamma}{\mathcal{WF}(E;E')[\Gamma]}$$

Second pruning property:

$$\frac{\mathcal{WF}(E;\ c := u : U;\ E')[\Gamma] \qquad c\ \text{does not occur in}\ E'\ \text{and}\ \Gamma}{\mathcal{WF}(E;E')[\Gamma]}$$

4.4.7 Co-inductive types

The implementation contains also co-inductive definitions, which are types inhabited by infinite objects. More information on co-inductive definitions can be found in *[Gimenez95][Gimenez98][GimenezCasteran05]*.

4.4.8 The Calculus of Inductive Constructions with impredicative Set

Coq can be used as a type checker for the 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,

Example

```
Fail Definition id: Set := forall X:Set,X->X.
    The command has indeed failed with message:
    The term "forall X : Set, X -> X" has type "Type"
    while it is expected to have type "Set" (universe inconsistency).
```

while it will type check, if one uses instead the `coqtop -impredicative-set` option..

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:

ProdImp

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

This extension has consequences on the inductive definitions which are allowed. In the impredicative system, one can build so-called *large inductive definitions* like the example of second-order existential quantifier (`exSet`).

There should be restrictions on the eliminations which can be performed on such definitions. The elimination rules in the impredicative system for sort Set become:

Set1

$$\frac{s \in \{\mathsf{Prop}, \mathsf{Set}\}}{[I : \mathsf{Set} | I \to s]}$$

Set2

$$\frac{I \text{ is a small inductive definition} \qquad s \in \{\mathsf{Type}(i)\}}{[I : \mathsf{Set} | I \to s]}$$

4.5 The Module System

The module system extends the Calculus of Inductive Constructions providing a convenient way to structure large developments as well as a means of massive abstraction.

4.5.1 Modules and module types

Access path. An access path is denoted by p and 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. A structure element is denoted by e and is either a definition of a constant, an assumption, a definition of an inductive, a definition of a module, an alias of a module or a module type abbreviation.

Structure expression. A structure expression is denoted by S and can be:

- an access path p
- a plain structure $\mathsf{Struct}\ e; \dots; e\ \mathsf{End}$
- a functor $\mathsf{Functor}(X : S)\ S'$, where X is a module variable, S and S' are structure expressions
- an application $S\ p$, where S is a structure expression and p an access path
- a refined structure $S\ \mathsf{with}\ p := p$ or $S\ \mathsf{with}\ p := t : T$ where S is a structure expression, p and p' are access paths, t is a term and T is the type of t.

Module definition. A module definition is written $\mathsf{Mod}(X : S\,[:= S'])$ and consists of a module variable X, a module type S which can be any structure expression and optionally a module implementation S' which can be any structure expression except a refined structure.

Module alias. A module alias is written $\mathsf{ModA}(X == p)$ and consists of a module variable X and a module path p.

Module type abbreviation. A module type abbreviation is written $\mathsf{ModType}(Y := S)$, where Y is an identifier and S is any structure expression .

4.5.2 Typing Modules

In order to introduce the typing system we first slightly extend the syntactic class of terms and environments given in section *The terms*. The environments, apart from definitions of constants and inductive types now also hold any other structure elements. Terms, apart from variables, constants and complex terms, include also access paths.

We also need additional typing judgments:

- $E[] \vdash \mathcal{WF}(S)$, denoting that a structure S is well-formed,
- $E[] \vdash p : S$, denoting that the module pointed by p has type S in environment E.
- $E[] \vdash S \longrightarrow \overline{S}$, denoting that a structure S is evaluated to a structure S in weak head normal form.
- $E[] \vdash S_1 <: S_2$, denoting that a structure S_1 is a subtype of a structure S_2.
- $E[] \vdash e_1 <: e_2$, denoting that a structure element e_1 is more precise than a structure element e_2.

The rules for forming structures are the following:

WF-STR

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

WF-FUN

$$\frac{E; \mathsf{Mod}(X : S)[] \vdash \mathcal{WF}(\overline{S'})}{E[] \vdash \mathcal{WF}(\mathsf{Functor}(X : S)\ S')}$$

Evaluation of structures to weak head normal form:

WEVAL-APP

$$\frac{\begin{array}{c} E[] \vdash S \longrightarrow \mathsf{Functor}(X : S_1)\ S_2 \qquad E[] \vdash S_1 \longrightarrow \overline{S_1} \\ E[] \vdash p : S_3 \qquad E[] \vdash S_3 <: \overline{S_1} \end{array}}{E[] \vdash S\ p \longrightarrow S_2\{p/X, t_1/p_1.c_1, \dots, t_n/p_n.c_n\}}$$

In the last rule, $\{t_1/p_1.c_1, \dots, t_n/p_n.c_n\}$ is the resulting substitution from the inlining mechanism. We substitute in S the inlined fields $p_i.c_i$ from $\mathsf{Mod}(X : S_1)$ by the corresponding delta- reduced term t_i in p.

WEVAL-WITH-MOD

$$\frac{\begin{array}{c} E[] \vdash S \longrightarrow \textsf{Struct}\ e_1;\ldots;e_i;\textsf{Mod}(X:S_1);e_{i+2};\ldots;e_n\ \textsf{End} \\ E;e_1;\ldots;e_i[] \vdash S_1 \longrightarrow \overline{S_1} \qquad\qquad E[] \vdash p : S_2 \\ E;e_1;\ldots;e_i[] \vdash S_2 <: \overline{S_1} \end{array}}{\begin{array}{c} E[] \vdash S\ \textsf{with}\ x := p \longrightarrow \\ \textsf{Struct}\ e_1;\ldots;e_i;\textsf{ModA}(X == p);e_{i+2}\{p/X\};\ldots;e_n\{p/X\}\ \textsf{End} \end{array}}$$

WEVAL-WITH-MOD-REC

$$\frac{\begin{array}{c} E[] \vdash S \longrightarrow \textsf{Struct}\ e_1;\ldots;e_i;\textsf{Mod}(X_1:S_1);e_{i+2};\ldots;e_n\ \textsf{End} \\ E;e_1;\ldots;e_i[] \vdash S_1\ \textsf{with}\ p := p_1 \longrightarrow \overline{S_2} \end{array}}{\begin{array}{c} E[] \vdash S\ \textsf{with}\ X_1.p := p_1 \longrightarrow \\ \textsf{Struct}\ e_1;\ldots;e_i;\textsf{Mod}(X:\overline{S_2});e_{i+2}\{p_1/X_1.p\};\ldots;e_n\{p_1/X_1.p\}\ \textsf{End} \end{array}}$$

WEVAL-WITH-DEF

$$\frac{\begin{array}{c} E[] \vdash S \longrightarrow \textsf{Struct}\ e_1;\ldots;e_i;\textsf{Assum}()(c:T_1);e_{i+2};\ldots;e_n\ \textsf{End} \\ E;e_1;\ldots;e_i[] \vdash Def()(c := t : T) <: \textsf{Assum}()(c:T_1) \end{array}}{\begin{array}{c} E[] \vdash S\ \textsf{with}\ c := t : T \longrightarrow \\ \textsf{Struct}\ e_1;\ldots;e_i;Def()(c := t : T);e_{i+2};\ldots;e_n\ \textsf{End} \end{array}}$$

WEVAL-WITH-DEF-REC

$$\frac{\begin{array}{c} E[] \vdash S \longrightarrow \textsf{Struct}\ e_1;\ldots;e_i;\textsf{Mod}(X_1:S_1);e_{i+2};\ldots;e_n\ \textsf{End} \\ E;e_1;\ldots;e_i[] \vdash S_1\ \textsf{with}\ p := p_1 \longrightarrow \overline{S_2} \end{array}}{\begin{array}{c} E[] \vdash S\ \textsf{with}\ X_1.p := t : T \longrightarrow \\ \textsf{Struct}\ e_1;\ldots;e_i;\textsf{Mod}(X:\overline{S_2});e_{i+2};\ldots;e_n\ \textsf{End} \end{array}}$$

WEVAL-PATH-MOD1

$$\frac{\begin{array}{c} E[] \vdash p \longrightarrow \textsf{Struct}\ e_1;\ldots;e_i;\textsf{Mod}(X:S\ [:= S_1]);e_{i+2};\ldots;e_nEnd \\ E;e_1;\ldots;e_i[] \vdash S \longrightarrow \overline{S} \end{array}}{E[] \vdash p.X \longrightarrow \overline{S}}$$

WEVAL-PATH-MOD2

$$\frac{\mathcal{WF}(E)[] \qquad \textsf{Mod}(X:S\ [:= S_1]) \in E \qquad E[] \vdash S \longrightarrow \overline{S}}{E[] \vdash X \longrightarrow \overline{S}}$$

WEVAL-PATH-ALIAS1

$$\frac{\begin{array}{c} E[] \vdash p \longrightarrow\ \textsf{Struct}\ e_1;\ldots;e_i;\textsf{ModA}(X == p_1);e_{i+2};\ldots;e_nEnd \\ E;e_1;\ldots;e_i[] \vdash p_1 \longrightarrow \overline{S} \end{array}}{E[] \vdash p.X \longrightarrow \overline{S}}$$

WEVAL-PATH-ALIAS2

$$\frac{\mathcal{WF}(E)[] \qquad \textsf{ModA}(X == p_1) \in E \qquad E[] \vdash p_1 \longrightarrow \overline{S}}{E[] \vdash X \longrightarrow \overline{S}}$$

WEVAL-PATH-TYPE1

$$\frac{\begin{array}{c} E[] \vdash p \longrightarrow\ \textsf{Struct}\ e_1;\ldots;e_i;\textsf{ModType}(Y := S);e_{i+2};\ldots;e_nEnd \\ E;e_1;\ldots;e_i[] \vdash S \longrightarrow \overline{S} \end{array}}{E[] \vdash p.Y \longrightarrow \overline{S}}$$

WEVAL-PATH-TYPE2

$$\frac{\mathcal{WF}(E)[] \qquad \mathsf{ModType}(Y := S) \in E \qquad E[] \vdash S \longrightarrow \overline{S}}{E[] \vdash Y \longrightarrow \overline{S}}$$

Rules for typing module:

MT-EVAL

$$\frac{E[] \vdash p \longrightarrow \overline{S}}{E[] \vdash p : \overline{S}}$$

MT-STR

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

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

- if $S \longrightarrow \mathsf{Struct}\ e_1; ...; e_n\ \mathsf{End}$ then $S/p = \mathsf{Struct}\ e_1/p; ...; e_n/p\ \mathsf{End}$ where e/p is defined as follows (note that opaque definitions are processed as assumptions):
  - $\mathsf{Def}()(c := t : T)/p = \mathsf{Def}()(c := t : T)$
  - $\mathsf{Assum}()(c : U)/p = \mathsf{Def}()(c := p.c : U)$
  - $\mathsf{Mod}(X : S)/p = \mathsf{ModA}(X == p.X)$
  - $\mathsf{ModA}(X == p')/p = \mathsf{ModA}(X == p')$
  - $\mathsf{Ind}[\Gamma_P](\Gamma_C := \Gamma_I)/p = \mathsf{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I)$
  - $\mathsf{Ind}_{p'}()[\Gamma_P](\Gamma_C := \Gamma_I)/p = \mathsf{Ind}_{p'}()[\Gamma_P](\Gamma_C := \Gamma_I)$
- if $S \longrightarrow \mathsf{Functor}(X : S')\ S''$ then $S/p = S$

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

The module subtyping rules:

MSUB-STR

$$\frac{\begin{array}{c} E; e_1; ...; e_n[] \vdash e_{\sigma(i)} <: e'_i \text{ for } i = 1..m \\ \sigma : \{1...m\} \to \{1...n\} \text{ injective} \end{array}}{E[] \vdash \mathsf{Struct}\ e_1; ...; e_n\ \mathsf{End} <: \mathsf{Struct}\ e'_1; ...; e'_m\ \mathsf{End}}$$

MSUB-FUN

$$\frac{E[] \vdash \overline{S'_1} <: \overline{S_1} \qquad E; \mathsf{Mod}(X : S'_1)[] \vdash \overline{S_2} <: \overline{S'_2}}{E[] \vdash \mathsf{Functor}(X : S_1)S_2 <: \mathsf{Functor}(X : S'_1)S'_2}$$

Structure element subtyping rules:

ASSUM-ASSUM

$$\frac{E[] \vdash T_1 \leq_{\beta\delta\iota\zeta\eta} T_2}{E[] \vdash \mathsf{Assum}()(c : T_1) <: \mathsf{Assum}()(c : T_2)}$$

DEF-ASSUM

$$\frac{E[] \vdash T_1 \leq_{\beta\delta\iota\zeta\eta} T_2}{E[] \vdash \mathsf{Def}()(c := t : T_1) <: \mathsf{Assum}()(c : T_2)}$$

ASSUM-DEF

$$\frac{E[] \vdash T_1 \leq_{\beta\delta\iota\zeta\eta} T_2 \qquad E[] \vdash c =_{\beta\delta\iota\zeta\eta} t_2}{E[] \vdash \mathsf{Assum}()(c : T_1) <: \mathsf{Def}()(c := t_2 : T_2)}$$

DEF-DEF

$$\frac{E[] \vdash T_1 \leq_{\beta\delta\iota\zeta\eta} T_2 \qquad E[] \vdash t_1 =_{\beta\delta\iota\zeta\eta} t_2}{E[] \vdash \mathsf{Def}()(c := t_1 : T_1) <: \mathsf{Def}()(c := t_2 : T_2)}$$

IND-IND

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

INDP-IND

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

INDP-INDP

$$\frac{\begin{array}{c} E[] \vdash \Gamma_P =_{\beta\delta\iota\zeta\eta} \Gamma'_P \qquad E[\Gamma_P] \vdash \Gamma_C =_{\beta\delta\iota\zeta\eta} \Gamma'_C \\ E[\Gamma_P;\Gamma_C] \vdash \Gamma_I =_{\beta\delta\iota\zeta\eta} \Gamma'_I \qquad E[] \vdash p =_{\beta\delta\iota\zeta\eta} p' \end{array}}{E[] \vdash \mathsf{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I) <: \mathsf{Ind}_{p'}()[\Gamma'_P](\Gamma'_C := \Gamma'_I)}$$

MOD-MOD

$$\frac{E[] \vdash S_1 <: S_2}{E[] \vdash \mathsf{Mod}(X : S_1) <: \mathsf{Mod}(X : S_2)}$$

ALIAS-MOD

$$\frac{E[] \vdash p : S_1 \qquad E[] \vdash S_1 <: S_2}{E[] \vdash \mathsf{ModA}(X == p) <: \mathsf{Mod}(X : S_2)}$$

MOD-ALIAS

$$\frac{E[] \vdash p : S_2 \qquad E[] \vdash S_1 <: S_2 \qquad E[] \vdash X =_{\beta\delta\iota\zeta\eta} p}{E[] \vdash \mathsf{Mod}(X : S_1) <: \mathsf{ModA}(X == p)}$$

ALIAS-ALIAS

$$\frac{E[] \vdash p_1 =_{\beta\delta\iota\zeta\eta} p_2}{E[] \vdash \mathsf{ModA}(X == p_1) <: \mathsf{ModA}(X == p_2)}$$

MODTYPE-MODTYPE

$$\frac{E[] \vdash S_1 <: S_2 \qquad E[] \vdash S_2 <: S_1}{E[] \vdash \mathsf{ModType}(Y := S_1) <: \mathsf{ModType}(Y := S_2)}$$

New environment formation rules

WF-MOD1

$$\frac{\mathcal{WF}(E)[] \qquad E[] \vdash \mathcal{WF}(S)}{WF(E;\mathsf{Mod}(X:S))[]}$$

WF-MOD2

$$\frac{E[] \vdash S_2 <: S_1 \qquad \mathcal{WF}(E)[] \qquad E[] \vdash \mathcal{WF}(S_1) \qquad E[] \vdash \mathcal{WF}(S_2)}{\mathcal{WF}(E;\mathsf{Mod}(X:S_1\,[:=S_2]))[]}$$

WF-ALIAS

$$\frac{\mathcal{WF}(E)[] \qquad E[] \vdash p:S}{\mathcal{WF}(E,\mathsf{ModA}(X==p))[]}$$

WF-MODTYPE

$$\frac{\mathcal{WF}(E)[] \qquad E[] \vdash \mathcal{WF}(S)}{\mathcal{WF}(E,\mathsf{ModType}(Y:=S))[]}$$

WF-IND

$$\frac{\begin{array}{c}\mathcal{WF}(E;\mathsf{Ind}\ [\Gamma_P]\,(\Gamma_C\ :=\ \Gamma_I))[]\\ E[] \vdash p:\ \mathsf{Struct}\ e_1;\ldots;e_n;\mathsf{Ind}\ [\Gamma'_P]\,(\Gamma'_C\ :=\ \Gamma'_I)\,;\ldots\ \mathsf{End}:\\ E[] \vdash \mathsf{Ind}\ [\Gamma'_P]\,(\Gamma'_C\ :=\ \Gamma'_I) <: \mathsf{Ind}\ [\Gamma_P]\,(\Gamma_C\ :=\ \Gamma_I)\end{array}}{\mathcal{WF}(E;\mathsf{Ind}_p()[\Gamma_P](\Gamma_C:=\Gamma_I))[]}$$

Component access rules

ACC-TYPE1

$$\frac{E[\Gamma] \vdash p:\ \mathsf{Struct}\ e_1;\ldots;e_i;\mathsf{Assum}()(c:T);\ldots\ \mathsf{End}}{E[\Gamma] \vdash p.c:T}$$

ACC-TYPE2

$$\frac{E[\Gamma] \vdash p:\ \mathsf{Struct}\ e_1;\ldots;e_i;\mathsf{Def}()(c:=t:T);\ldots\ \mathsf{End}}{E[\Gamma] \vdash p.c:T}$$

Notice that the following rule extends the delta rule defined in section *Conversion rules*

ACC-DELTA

$$\frac{E[\Gamma] \vdash p:\ \mathsf{Struct}\ e_1;\ldots;e_i;\mathsf{Def}()(c:=t:U);\ldots\ \mathsf{End}}{E[\Gamma] \vdash p.c \rhd_\delta t}$$

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

ACC-IND1

$$\frac{E[\Gamma] \vdash p:\ \mathsf{Struct}\ e_1;\ldots;e_i;\mathsf{Ind}\ [\Gamma_P]\,(\Gamma_C\ :=\ \Gamma_I)\,;\ldots\ \mathsf{End}}{E[\Gamma] \vdash p.I_j:(p_1:P_1)\ldots(p_r:P_r)A_j}$$

ACC-IND2

$$\frac{E[\Gamma] \vdash p : \ \mathsf{Struct}\ e_1; \ldots; e_i; \mathsf{Ind}\ [\Gamma_P]\,(\Gamma_C\ := \ \Gamma_I)\,; \ldots\ \mathsf{End}}{E[\Gamma] \vdash p.c_m : (p_1 : P_1) \ldots (p_r : P_r) C_m I_j (I_j\ p_1 \ldots p_r)_{j=1\ldots k}}$$

ACC-INDP1

$$\frac{E[] \vdash p : \ \mathsf{Struct}\ e_1; \ldots; e_i; \mathsf{Ind}_{p'}()[\Gamma_P](\Gamma_C := \Gamma_I); \ldots\ \mathsf{End}}{E[] \vdash p.I_i \rhd_\delta p'.I_i}$$

ACC-INDP2

$$\frac{E[] \vdash p : \ \mathsf{Struct}\ e_1; \ldots; e_i; \mathsf{Ind}_{p'}()[\Gamma_P](\Gamma_C := \Gamma_I); \ldots\ \mathsf{End}}{E[] \vdash p.c_i \rhd_\delta p'.c_i}$$

CHAPTER

FIVE

THE PROOF ENGINE

5.1 Vernacular commands

5.1.1 Displaying

Command: `Print` *qualid*

This command displays on the screen information about the declared or defined object referred by *qualid*.

Error messages:

Error: *qualid* `not a defined object.`

Error: `Universe instance should have length` *num*`.`

Error: `This object does not support universe names.`

Variant: `Print Term` *qualid*

This is a synonym of *Print qualid* when *qualid* denotes a global constant.

Variant: `Print` `Term`? *qualid*`@`*name*

This locally renames the polymorphic universes of *qualid*. An underscore means the usual name is printed.

Command: `About` *qualid*

This displays various information about the object denoted by *qualid*: its kind (module, constant, assumption, inductive, constructor, abbreviation, …), long name, type, implicit arguments and argument scopes. It does not print the body of definitions or proofs.

Variant: `About` *qualid*`@`*name*

This locally renames the polymorphic universes of *qualid*. An underscore means the usual name is printed.

Command: `Print All`

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

Variant: `Inspect` *num*

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

Variant: `Print Section` *ident*

The name *ident* should correspond to a currently open section, this command displays the objects defined since the beginning of this section.

5.1.2 Flags, Options and Tables

Coq has many settings to control its behavior. Setting types include flags, options and tables:

- A `flag` has a boolean value, such as *Asymmetric Patterns*.
- An `option` generally has a numeric or string value, such as *Firstorder Depth*.
- A `table` contains a set of strings or qualids.
- In addition, some commands provide settings, such as *Extraction Language*.

Flags, options and tables are identified by a series of identifiers, each with an initial capital letter.

Command: `Local` | `Global` | `Export` ? `Set` *flag*

Sets *flag* on. Scoping qualifiers are described *here*.

Command: `Local` | `Global` | `Export` ? `Unset` *flag*

Sets *flag* off. Scoping qualifiers are described *here*.

Command: `Test` *flag*

Prints the current value of *flag*.

Command: `Local` | `Global` | `Export` ? `Set` *option* *num* | *string*

Sets *option* to the specified value. Scoping qualifiers are described *here*.

Command: `Local` | `Global` | `Export` ? `Unset` *option*

Sets *option* to its default value. Scoping qualifiers are described *here*.

Command: `Test` *option*

Prints the current value of *option*.

Command: `Print Options`

Prints the current value of all flags and options, and the names of all tables.

Command: `Add` *table* *string* | *qualid*

Adds the specified value to *table*.

Command: `Remove` *table* *string* | *qualid*

Removes the specified value from *table*.

Command: `Test` *table* `for` *string* | *qualid*

Reports whether *table* contains the specified value.

Command: `Print Table` *table*

Prints the values in *table*.

Command: `Test` *table*

A synonym for *Print Table @table*.

Command: `Print Tables`

A synonym for *Print Options*.

Scope qualifiers for `Set` and `Unset`

`Local` | `Global` | `Export` ?

Flag and option settings can be global in scope or local to nested scopes created by *Module* and *Section* commands. There are four alternatives:

- no qualifier: the original setting is *not* restored at the end of the current module or section.

- **Local**: the setting is applied within the current scope. The original value of the option or flag is restored at the end of the current module or section.
- **Global**: similar to no qualifier, the original setting is *not* restored at the end of the current module or section. In addition, if the value is set in a file, then *Require*-ing the file sets the option.
- **Export**: similar to **Local**, the original value of the option or flag is restored at the end of the current module or section. In addition, if the value is set in a file, then *Import*-ing the file sets the option.

Newly opened scopes inherit the current settings.

5.1.3 Requests to the environment

Command: `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.

Variant: *selector*: `Check` *term*

This variant specifies on which subgoal to perform typing (see Section *Invocation of tactics*).

Command: `Eval` *redexpr* `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 *Performing computations*.

Command: `Compute` *term*

This command performs a call-by-value evaluation of term by using the bytecode-based virtual machine. It is a shortcut for `Eval vm_compute in` *term*.

See also:

Section *Performing computations*.

Command: `Print Assumptions` *qualid*

This commands display all the assumptions (axioms, parameters and variables) a theorem or definition depends on. Especially, it informs on the assumptions with respect to which the validity of a theorem relies.

Variant: `Print Opaque Dependencies` *qualid*

Displays the set of opaque constants *qualid* relies on in addition to the assumptions.

Variant: `Print Transparent Dependencies` *qualid*

Displays the set of transparent constants *qualid* relies on in addition to the assumptions.

Variant: `Print All Dependencies` *qualid*

Displays all assumptions and constants *qualid* relies on.

Command: `Search` *qualid*

This command displays the name and type of all objects (hypothesis of the current goal, 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: `The reference` *qualid* `was not found in the current environment.`

There is no constant in the environment named qualid.

Variant: `Search` *string*

If *string* is a valid identifier, this command displays the name and type of all objects (theorems, axioms, etc) of the current context whose name contains string. If string is a notation's string denoting some reference *qualid* (referred to by its main symbol as in "+" or by its notation's string as in "_ + _" or "_ 'U' _", see Section *Notations*), the command works like Search *qualid*.

Variant: Search *string*%*ident*

The string string must be a notation or the main symbol of a notation which is then interpreted in the scope bound to the delimiting key *ident* (see Section *Local interpretation rules for notations*).

Variant: Search *term_pattern*

This searches for all statements or types of definition that contains a subterm that matches the pattern term_pattern (holes of the pattern are either denoted by _ or by ?*ident* when non linear patterns are expected).

Variant: Search -? *term_pattern_string*+

where *term_pattern_string* is a term_pattern, a string, or a string followed by a scope delimiting key %key. This generalization of Search searches for all objects whose statement or type contains a subterm matching *term_pattern* (or *qualid* if *string* is the notation for a reference qualid) and whose name contains all string of the request that correspond to valid identifiers. If a term_pattern or a string is prefixed by -, the search excludes the objects that mention that term_pattern or that string.

Variant: Search -? *term_pattern_string*+ inside *qualid*+

This restricts the search to constructions defined in the modules named by the given qualid sequence.

Variant: Search -? *term_pattern_string*+ outside *qualid*+

This restricts the search to constructions not defined in the modules named by the given qualid sequence.

Variant: *selector*: Search -? *term_pattern_string*+

This specifies the goal on which to search hypothesis (see Section *Invocation of tactics*). By default the 1st goal is searched. This variant can be combined with other variants presented here.

Example

```
Require Import ZArith.

Search Z.mul Z.add "distr".
    Z.mul_add_distr_l: forall n m p : Z, (n * (m + p))%Z = (n * m + n * p)%Z
    Z.mul_add_distr_r: forall n m p : Z, ((n + m) * p)%Z = (n * p + m * p)%Z
    fast_Zmult_plus_distr_l:
      forall (n m p : Z) (P : Z -> Prop),
      P (n * p + m * p)%Z -> P ((n + m) * p)%Z

Search "+"%Z "*"%Z "distr" -positive -Prop.
    Z.mul_add_distr_l: forall n m p : Z, (n * (m + p))%Z = (n * m + n * p)%Z
    Z.mul_add_distr_r: forall n m p : Z, ((n + m) * p)%Z = (n * p + m * p)%Z

Search (?x * _ + ?x * _)%Z outside OmegaLemmas.
    Z.mul_add_distr_l: forall n m p : Z, (n * (m + p))%Z = (n * m + n * p)%Z
```

Variant: SearchAbout

Deprecated since version 8.5.

Up to Coq version 8.4, *Search* had the behavior of current *SearchHead* and the behavior of current *Search* was obtained with command *SearchAbout*. For compatibility, the deprecated name *SearchAbout* can still be used as a synonym of *Search*. For compatibility, the list of objects to search when using *SearchAbout* may also be enclosed by optional `[ ]` delimiters.

Command: `SearchHead` *term*

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

Example

```
SearchHead le.
    le_n: forall n : nat, n <= n
    le_0_n: forall n : nat, 0 <= n
    le_S: forall n m : nat, n <= m -> n <= S m
    le_pred: forall n m : nat, n <= m -> Nat.pred n <= Nat.pred m
    le_n_S: forall n m : nat, n <= m -> S n <= S m
    le_S_n: forall n m : nat, S n <= S m -> n <= m

SearchHead (@eq bool).
    andb_true_intro:
      forall b1 b2 : bool, b1 = true /\ b2 = true -> (b1 && b2)%bool = true
```

Variant: `SearchHead` *term* `inside` *qualid*+

This restricts the search to constructions defined in the modules named by the given `qualid` sequence.

Variant: `SearchHead` *term* `outside` *qualid*+

This restricts the search to constructions not defined in the modules named by the given `qualid` sequence.

Error: `Module/section` *qualid* `not found.`

No module *qualid* has been required (see Section *Compiled files*).

Variant: *selector*: `SearchHead` *term*

This specifies the goal on which to search hypothesis (see Section *Invocation of tactics*). By default the 1st goal is searched. This variant can be combined with other variants presented here.

Note: Up to Coq version 8.4, `SearchHead` was named `Search`.

Command: `SearchPattern` *term*

This command displays the name and type of all hypothesis of the current goal (if any) and theorems of the current context whose statement's conclusion or last hypothesis and conclusion matches the expressionterm where holes in the latter are denoted by `_`. It is a variant of `Search` *term_pattern* that does not look for subterms but searches for statements whose conclusion has exactly the expected form, or whose statement finishes by the given series of hypothesis/conclusion.

Example

```
Require Import Arith.
```

```
SearchPattern (_ + _ = _ + _).
    Nat.add_comm: forall n m : nat, n + m = m + n
    plus_Snm_nSm: forall n m : nat, S n + m = n + S m
    Nat.add_succ_comm: forall n m : nat, S n + m = n + S m
    Nat.add_shuffle3: 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)
    Nat.add_assoc: forall n m p : nat, n + (m + p) = n + m + p
    Nat.add_shuffle0: forall n m p : nat, n + m + p = n + p + m
    f_equal2_plus:
      forall x1 y1 x2 y2 : nat, x1 = y1 -> x2 = y2 -> x1 + x2 = y1 + y2
    Nat.add_shuffle2: forall n m p q : nat, n + m + (p + q) = n + q + (m + p)
    Nat.add_shuffle1: forall n m p q : nat, n + m + (p + q) = n + p + (m + q)

SearchPattern (nat -> bool).
    Nat.odd: nat -> bool
    Init.Nat.odd: nat -> bool
    Nat.even: nat -> bool
    Init.Nat.even: nat -> bool
    Init.Nat.testbit: nat -> nat -> bool
    Nat.leb: nat -> nat -> bool
    Nat.eqb: nat -> nat -> bool
    Init.Nat.eqb: nat -> nat -> bool
    Nat.ltb: nat -> nat -> bool
    Nat.testbit: nat -> nat -> bool
    Init.Nat.leb: nat -> nat -> bool
    Init.Nat.ltb: nat -> nat -> bool
    BinNat.N.testbit_nat: BinNums.N -> nat -> bool
    BinPosDef.Pos.testbit_nat: BinNums.positive -> nat -> bool
    BinPos.Pos.testbit_nat: BinNums.positive -> nat -> bool
    BinNatDef.N.testbit_nat: BinNums.N -> nat -> bool

SearchPattern (forall l : list _, _ l l).
    List.incl_refl: forall (A : Type) (l : list A), List.incl l l
    List.lel_refl: forall (A : Type) (l : list A), List.lel l l
```

Patterns need not be linear: you can express that the same expression must occur in two places by using pattern variables `?ident`.

Example

```
SearchPattern (?X1 + _ = _ + ?X1).
    Nat.add_comm: forall n m : nat, n + m = m + n
```

Variant: `SearchPattern` *term* `inside` *qualid*+

This restricts the search to constructions defined in the modules named by the given `qualid` sequence.

Variant: `SearchPattern` *term* `outside` *qualid*+

This restricts the search to constructions not defined in the modules named by the given `qualid` sequence.

Variant: *selector*: `SearchPattern` *term*

This specifies the goal on which to search hypothesis (see Section *Invocation of tactics*). By default the 1st goal is searched. This variant can be combined with other variants presented here.

Command: `SearchRewrite` *`term`*

This command displays the name and type of all hypothesis of the current goal (if any) and 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 "_".

Example

```
Require Import Arith.

SearchRewrite (_ + _ + _).
    Nat.add_shuffle0: forall n m p : nat, n + m + p = n + p + m
    plus_assoc_reverse: forall n m p : nat, n + m + p = n + (m + p)
    Nat.add_assoc: forall n m p : nat, n + (m + p) = n + m + p
    Nat.add_shuffle1: forall n m p q : nat, n + m + (p + q) = n + p + (m + q)
    Nat.add_shuffle2: forall n m p q : nat, n + m + (p + q) = n + q + (m + p)
    Nat.add_carry_div2:
      forall (a b : nat) (c0 : bool),
      (a + b + Nat.b2n c0) / 2 =
      a / 2 + b / 2 +
      Nat.b2n
        (Nat.testbit a 0 && Nat.testbit b 0
         || c0 && (Nat.testbit a 0 || Nat.testbit b 0))
```

Variant: `SearchRewrite` *`term`* `inside` *`qualid`*+

This restricts the search to constructions defined in the modules named by the given `qualid` sequence.

Variant: `SearchRewrite` *`term`* `outside` *`qualid`*+

This restricts the search to constructions not defined in the modules named by the given `qualid` sequence.

Variant: *`selector`*`: SearchRewrite` *`term`*

This specifies the goal on which to search hypothesis (see Section *Invocation of tactics*). By default the 1st goal is searched. This variant can be combined with other variants presented here.

Note:

Table: `Search Blacklist` *`string`*

Specifies a set of strings used to exclude lemmas from the results of *`Search`*, *`SearchHead`*, *`SearchPattern`* and *`SearchRewrite`* queries. A lemma whose fully-qualified name contains any of the strings will be excluded from the search results. The default blacklisted substrings are `_subterm`, `_subproof` and `Private_`.

Use the *`Add @table`* and *`Remove @table`* commands to update the set of blacklisted strings.

Command: `Locate` *`qualid`*

This command displays the full name of objects whose name is a prefix of the qualified identifier *`qualid`*, and consequently the Coq module in which they are defined. It searches for objects from the different qualified namespaces of Coq: terms, modules, Ltac, etc.

Example

```
Locate nat.
    Inductive Coq.Init.Datatypes.nat

Locate Datatypes.O.
    Constructor Coq.Init.Datatypes.O
      (shorter name to refer to it in current context is O)

Locate Init.Datatypes.O.
    Constructor Coq.Init.Datatypes.O
      (shorter name to refer to it in current context is O)

Locate Coq.Init.Datatypes.O.
    Constructor Coq.Init.Datatypes.O
      (shorter name to refer to it in current context is O)

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

Variant: `Locate Term` *`qualid`*

As Locate but restricted to terms.

Variant: `Locate Module` *`qualid`*

As Locate but restricted to modules.

Variant: `Locate Ltac` *`qualid`*

As Locate but restricted to tactics.

See also:

Section *Locating notations*

5.1.4 Printing flags

Flag: `Fast Name Printing`

When turned on, Coq uses an asymptotically faster algorithm for the generation of unambiguous names of bound variables while printing terms. While faster, it is also less clever and results in a typically less elegant display, e.g. it will generate more names rather than reusing certain names across subterms. This flag is not enabled by default, because as Ltac observes bound names, turning it on can break existing proof scripts.

5.1.5 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.

Command: `Load` *`ident`*

This command loads the file named `ident`.v, searching successively in each of the directories specified in the *loadpath*. (see Section *Libraries and filesystem*)

Files loaded this way cannot leave proofs open, and the `Load` command cannot be used inside a proof either.

Variant: `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`.

Variant: `Load Verbose` *`ident`*
Variant: `Load Verbose` *`string`*

Display, while loading, the answers of Coq to each command (including tactics) contained in the loaded file.

See also:

Section *Controlling display*.

Error: **`Can’t find file`** *`ident`* **`on loadpath.`**

Error: **`Load is not supported inside proofs.`**

Error: **`Files processed by Load cannot leave open proofs.`**

5.1.6 Compiled files

This section describes the commands used to load compiled files (see Chapter *The Coq commands* for documentation on how to compile a file). A compiled file is a particular case of module called *library file*.

Command: `Require` *`qualid`*

This command looks in the loadpath for a file containing module *`qualid`* and adds the corresponding module to the environment of Coq. As library files have dependencies in other library files, the command *`Require qualid`* recursively requires all library files the module qualid depends on and adds the corresponding modules to the environment of Coq too. Coq assumes that the compiled files have been produced by a valid Coq compiler and their contents are then not replayed nor rechecked.

To locate the file in the file system, *`qualid`* is decomposed under the form `dirpath.`*`ident`* and the file *`ident`*`.vo` is searched in the physical directory of the file system that is mapped in Coq loadpath to the logical path dirpath (see Section *Libraries and filesystem*). The mapping between physical directories and logical names at the time of requiring the file must be consistent with the mapping used to compile the file. If several files match, one of them is picked in an unspecified fashion.

Variant: `Require Import` *`qualid`*

This loads and declares the module *`qualid`* and its dependencies then imports the contents of *`qualid`* as described *here*. It does not import the modules on which qualid depends unless these modules were themselves required in module *`qualid`* using *`Require Export`*, as described below, or recursively required through a sequence of *`Require Export`*. If the module required has already been loaded, *`Require Import qualid`* simply imports it, as *`Import qualid`* would.

Variant: `Require Export` *`qualid`*

This command acts as *`Require Import qualid`*, but if a further module, say `A`, contains a command *`Require Export`* `B`, then the command *`Require Import`* `A` also imports the module `B`.

Variant: `Require` `Import` | `Export` *`qualid`*+

This loads the modules named by the *`qualid`* sequence and their recursive dependencies. If `Import` or `Export` is given, it also imports these modules and all the recursive dependencies that were marked or transitively marked as `Export`.

Variant: `From` *`dirpath`* `Require` *`qualid`*

This command acts as *`Require`*, but picks any library whose absolute name is of the form *`dirpath`*`.`*`dirpath`*`’.`*`qualid`* for some *`dirpath`*`’`. This is useful to ensure that the *`qualid`* library comes from a given package by making explicit its absolute root.

Error: **`Cannot load qualid: no physical path bound to dirpath.`**

Error: `Cannot find library foo in loadpath.`
The command did not find the file foo.vo. Either foo.v exists but is not compiled or foo.vo is in a directory which is not in your LoadPath (see Section *Libraries and filesystem*).

Error: `Compiled library` *`ident`*`.vo makes inconsistent assumptions over library qualid.`
The command tried to load library file *`ident`*.vo that depends on some specific version of library *`qualid`* which is not the one already loaded in the current Coq session. Probably *`ident`*`.v` was not properly recompiled with the last version of the file containing module *`qualid`*.

Error: `Bad magic number.`
The file *`ident`*`.vo` was found but either it is not a Coq compiled module, or it was compiled with an incompatible version of Coq.

Error: `The file :n:`*`ident`*`.vo` contains library dirpath and not library dirpath'.`
The library file *`dirpath`*' is indirectly required by the `Require` command but it is bound in the current loadpath to the file *`ident`*`.vo` which was bound to a different library name `dirpath` at the time it was compiled.

Error: `Require is not allowed inside a module or a module type.`
This command is not allowed inside a module or a module type being defined. It is meant to describe a dependency between compilation units. Note however that the commands `Import` and `Export` alone can be used inside modules (see Section *Import*).

See also:

Chapter *The Coq commands*

Command: `Print Libraries`
This command displays the list of library files loaded in the current Coq session.

Command: `Declare ML Module` *`string`*+
This commands loads the OCaml compiled files with names given by the *`string`* sequence (dynamic link). It is mainly used to load tactics dynamically. The files are searched into the current OCaml loadpath (see the command *`Add ML Path`*). Loading of OCaml files is only possible under the bytecode version of `coqtop` (i.e. `coqtop` called with option `-byte`, see chapter *The Coq commands*), or when Coq has been compiled with a version of OCaml that supports native Dynlink (3.11).

Variant: `Local Declare ML Module` *`string`*+
This variant is not exported to the modules that import the module where they occur, even if outside a section.

Error: `File not found on loadpath:` *`string`*`.`

Error: `Loading of ML object file forbidden in a native Coq.`

Command: `Print ML Modules`
This prints the name of all OCaml modules loaded with *`Declare ML Module`*. To know from where these module were loaded, the user should use the command *`Locate File`*.

5.1.7 Loadpath

Loadpaths are preferably managed using Coq command line options (see Section `libraries-and-filesystem`) but there remain vernacular commands to manage them for practical purposes. Such commands are only meant to be issued in the toplevel, and using them in source files is discouraged.

Command: `Pwd`
This command displays the current working directory.

Command: `Cd` *`string`*

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

Variant: `Cd`

Is equivalent to Pwd.

Command: `Add LoadPath` *`string`* `as` *`dirpath`*

This command is equivalent to the command line option `-Q` *`string dirpath`*. It adds the physical directory string to the current Coq loadpath and maps it to the logical directory dirpath.

Variant: `Add LoadPath` *`string`*

Performs as `Add LoadPath` *`string dirpath`* but for the empty directory path.

Command: `Add Rec LoadPath` *`string`* `as` *`dirpath`*

This command is equivalent to the command line option `-R` *`string dirpath`*. It adds the physical directory string and all its subdirectories to the current Coq loadpath.

Variant: `Add Rec LoadPath` *`string`*

Works as `Add Rec LoadPath` *`string`* `as` *`dirpath`* but for the empty logical directory path.

Command: `Remove LoadPath` *`string`*

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

Command: `Print LoadPath`

This command displays the current Coq loadpath.

Variant: `Print LoadPath` *`dirpath`*

Works as *`Print LoadPath`* but displays only the paths that extend the *`dirpath`* prefix.

Command: `Add ML Path` *`string`*

This command adds the path *`string`* to the current OCaml loadpath (see the command `Declare ML Module`` in Section *Compiled files*).

Command: `Add Rec ML Path` *`string`*

This command adds the directory *`string`* and all its subdirectories to the current OCaml loadpath (see the command *`Declare ML Module`*).

Command: `Print ML Path` *`string`*

This command displays the current OCaml loadpath. This command makes sense only under the bytecode version of `coqtop`, i.e. using option `-byte` (see the command Declare ML Module in Section *Compiled files*).

Command: `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.

Command: `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`*.

5.1.8 Backtracking

The backtracking commands described in this section can only be used interactively, they cannot be part of a vernacular file loaded via `Load` or compiled by `coqc`.

Command: `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: *`ident`*`: no such entry.`

Variant: `Reset Initial`

Goes back to the initial state, just after the start of the interactive session.

Command: `Back`

This command undoes all the effects of the last vernacular command. Commands read from a vernacular file via a *Load* are considered as a single command. Proof management commands are also handled by this command (see Chapter *Proof handling*). For that, Back may have to undo more than one command in order to reach a state where the proof management information is available. For instance, when the last command is a *Qed*, the management information about the closed proof has been discarded. In this case, *Back* will then undo all the proof steps up to the statement of this proof.

Variant: `Back` *num*

Undo *num* vernacular commands. As for Back, some extra commands may be undone in order to reach an adequate state. For instance Back *num* will not re-enter a closed proof, but rather go just before that proof.

Error: `Invalid backtrack.`

The user wants to undo more commands than available in the history.

Command: `BackTo` *num*

This command brings back the system to the state labeled *num*, forgetting the effect of all commands executed after this state. The state label is an integer which grows after each successful command. It is displayed in the prompt when in -emacs mode. Just as *Back* (see above), the *BackTo* command now handles proof states. For that, it may have to undo some extra commands and end on a state *num* *num* if necessary.

5.1.9 Quitting and debugging

Command: `Quit`

This command permits to quit Coq.

Command: `Drop`

This is used mostly as a debug facility by Coq's implementers and does not concern the casual user. This command permits to leave Coq temporarily and enter the OCaml toplevel. The OCaml command:

```
#use "include";;
```

adds the right loadpaths and loads some toplevel printers for all abstract types of Coq- section_path, identifiers, terms, judgments, …. You can also use the file base_include instead, that loads only the pretty-printers for section_paths and identifiers. You can return back to Coq with the command:

```
go();;
```

Warning:

1. It only works with the bytecode version of Coq (i.e. `coqtop.byte`, see Section `interactive-use`).
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 `customization-by-environment-variables`).

Command: `Time` *command*

This command executes the vernacular command *command* and displays the time needed to execute it.

Command: `Redirect` *`string`* *`command`*

This command executes the vernacular command *`command`*, redirecting its output to ”*`string`*.out”.

Command: `Timeout` *`num`* *`command`*

This command executes the vernacular command *`command`*. If the command has not terminated after the time specified by the *`num`* (time expressed in seconds), then it is interrupted and an error message is displayed.

Option: `Default Timeout` *`num`*

This option controls a default timeout for subsequent commands, as if they were passed to a *`Timeout`* command. Commands already starting by a *`Timeout`* are unaffected.

Command: `Fail` *`command`*

For debugging scripts, sometimes it is desirable to know whether a command or a tactic fails. If the given *`command`* fails, then `Fail` *`command`* succeeds (excepts in the case of critical errors, like a ”stack overflow”), without changing the proof state, and in interactive mode, the system prints a message confirming the failure.

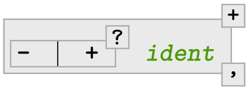
Error: `The command has not failed!`

If the given *`command`* succeeds, then `Fail` *`command`* fails with this error message.

5.1.10 Controlling display

Flag: `Silent`

This flag controls the normal displaying.

Option: `Warnings` `"`  `"`

This option configures the display of warnings. It is experimental, and expects, between quotes, a comma-separated list of warning names or categories. Adding - in front of a warning or category disables it, adding + makes it an error. It is possible to use the special categories all and default, the latter containing the warnings enabled by default. The flags are interpreted from left to right, so in case of an overlap, the flags on the right have higher priority, meaning that `A,-A` is equivalent to `-A`.

Flag: `Search Output Name Only`

This flag restricts the output of search commands to identifier names; turning it on causes invocations of *`Search`*, *`SearchHead`*, *`SearchPattern`*, *`SearchRewrite`* etc. to omit types from their output, printing only identifiers.

Option: `Printing Width` *`num`*

This command sets which left-aligned part of the width of the screen is used for display. At the time of writing this documentation, the default value is 78.

Option: `Printing Depth` *`num`*

This option controls the nesting depth of the formatter used for pretty- printing. Beyond this depth, display of subterms is replaced by dots. At the time of writing this documentation, the default value is 50.

Flag: `Printing Compact Contexts`

This flag controls the compact display mode for goals contexts. When on, the printer tries to reduce the vertical size of goals contexts by putting several variables (even if of different types) on the same line provided it does not exceed the printing width (see *`Printing Width`*). At the time of writing this documentation, it is off by default.

Flag: `Printing Unfocused`

This flag controls whether unfocused goals are displayed. Such goals are created by focusing other goals with bullets (see *Bullets* or *curly braces*). It is off by default.

Flag: `Printing Dependent Evars Line`

This flag controls the printing of the "(dependent evars: …)" information after each tactic. The information is used by the Prooftree tool in Proof General. (https://askra.de/software/prooftree)

5.1.11 Controlling the reduction strategies and the conversion algorithm

Coq provides reduction strategies that the tactics can invoke and two different algorithms to check the convertibility of types. The first conversion algorithm lazily compares applicative terms while the other is a brute-force but efficient algorithm that first normalizes the terms before comparing them. The second algorithm is based on a bytecode representation of terms similar to the bytecode representation used in the ZINC virtual machine *[Ler90]*. It is especially useful for intensive computation of algebraic values, such as numbers, and for reflection-based tactics. The commands to fine- tune the reduction strategies and the lazy conversion algorithm are described first.

Command: `Opaque` *`qualid`*+

This command has an effect on unfoldable constants, i.e. on constants defined by *`Definition`* or *`Let`* (with an explicit body), or by a command assimilated to a definition such as *`Fixpoint`*, *`Program Definition`*, etc, or by a proof ended by *`Defined`*. The command tells not to unfold the constants in the *`qualid`* sequence in tactics using δ-conversion (unfolding a constant is replacing it by its definition).

`Opaque` has also an effect on the conversion algorithm of Coq, telling it to delay the unfolding of a constant as much as possible when Coq has to check the conversion (see Section *Conversion rules*) of two distinct applied constants.

Variant: `Global Opaque` *`qualid`*+

The scope of *`Opaque`* is limited to the current section, or current file, unless the variant *`Global Opaque`* is used.

See also:

Sections *Performing computations*, *Automating*, *Switching on/off the proof editing mode*

Error: **`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.

Command: `Transparent` *`qualid`*+

This command is the converse of *`Opaque`* and it applies on unfoldable constants to restore their unfoldability after an Opaque command.

Note in particular that constants defined by a proof ended by Qed are not unfoldable and Transparent has no effect on them. This is to keep with the usual mathematical practice of *proof irrelevance*: what matters in a mathematical development is the sequence of lemma statements, not their actual proofs. This distinguishes lemmas from the usual defined constants, whose actual values are of course relevant in general.

Variant: `Global Transparent` *`qualid`*+

The scope of Transparent is limited to the current section, or current file, unless the variant *`Global Transparent`* is used.

Error: **`The reference`** *`qualid`* **`was not found in the current environment.`**

There is no constant referred by *`qualid`* in the environment.

See also:

Sections *Performing computations*, *Automating*, *Switching on/off the proof editing mode*

Command: `Strategy` *`level`* `[` *`qualid`*$^+$ `]`

This command generalizes the behavior of Opaque and Transparent commands. It is used to fine-tune the strategy for unfolding constants, both at the tactic level and at the kernel level. This command associates a level to the qualified names in the *`qualid`* sequence. Whenever two expressions with two distinct head constants are compared (for instance, this comparison can be triggered by a type cast), the one with lower level is expanded first. In case of a tie, the second one (appearing in the cast type) is expanded.

```
level ::= opaque | num | expand
```

Levels can be one of the following (higher to lower):

- `opaque` : level of opaque constants. They cannot be expanded by tactics (behaves like $+\infty$, see next item).
- *`num`* : levels indexed by an integer. Level 0 corresponds to the default behavior, which corresponds to transparent constants. This level can also be referred to as transparent. Negative levels correspond to constants to be expanded before normal transparent constants, while positive levels correspond to constants to be expanded after normal transparent constants.
- `expand` : level of constants that should be expanded first (behaves like $-\infty$)

Variant: `Local Strategy` *`level`* `[` *`qualid`*$^+$ `]`

These directives survive section and module closure, unless the command is prefixed by `Local`. In the latter case, the behavior regarding sections and modules is the same as for the *`Transparent`* and *`Opaque`* commands.

Command: `Print Strategy` *`qualid`*

This command prints the strategy currently associated to *`qualid`*. It fails if *`qualid`* is not an unfoldable reference, that is, neither a variable nor a constant.

Error: `The reference is not unfoldable.`

Variant: `Print Strategies`

Print all the currently non-transparent strategies.

Command: `Declare Reduction` *`ident`* `:=` *`redexpr`*

This command allows giving a short name to a reduction expression, for instance `lazy beta delta [foo bar]`. This short name can then be used in `Eval` *`ident`* `in` or `eval` directives. This command accepts the `Local` modifier, for discarding this reduction name at the end of the file or module. For the moment, the name is not qualified. In particular declaring the same name in several modules or in several functor applications will be rejected if these declarations are not local. The name *`ident`* cannot be used directly as an Ltac tactic, but nothing prevents the user from also performing a `Ltac` *`ident`* `:=` *`redexpr`*.

See also:

Performing computations

5.1.12 Controlling the locality of commands

Command: `Local` *`command`*
Command: `Global` *`command`*

Some commands support a Local or Global prefix modifier to control the scope of their effect. There are four kinds of commands:

- Commands whose default is to extend their effect both outside the section and the module or library file they occur in. For these commands, the Local modifier limits the effect of the command to the current section or module it occurs in. As an example, the *Coercion* and *Strategy* commands belong to this category.

- Commands whose default behavior is to stop their effect at the end of the section they occur in but to extend their effect outside the module or library file they occur in. For these commands, the Local modifier limits the effect of the command to the current module if the command does not occur in a section and the Global modifier extends the effect outside the current sections and current module if the command occurs in a section. As an example, the *Arguments*, *Ltac* or *Notation* commands belong to this category. Notice that a subclass of these commands do not support extension of their scope outside sections at all and the Global modifier is not applicable to them.
- Commands whose default behavior is to stop their effect at the end of the section or module they occur in. For these commands, the `Global` modifier extends their effect outside the sections and modules they occur in. The *Transparent* and *Opaque* (see Section *Controlling the reduction strategies and the conversion algorithm*) commands belong to this category.
- Commands whose default behavior is to extend their effect outside sections but not outside modules when they occur in a section and to extend their effect outside the module or library file they occur in when no section contains them. For these commands, the Local modifier limits the effect to the current section or module while the Global modifier extends the effect outside the module even when the command occurs in a section. The *Set* and *Unset* commands belong to this category.

5.1.13 Controlling Typing Flags

Flag: `Guard Checking`

This flag can be used to enable/disable the guard checking of fixpoints. Warning: this can break the consistency of the system, use at your own risk. Decreasing argument can still be specified: the decrease is not checked anymore but it still affects the reduction of the term. Unchecked fixpoints are printed by *Print Assumptions*.

Flag: `Positivity Checking`

This flag can be used to enable/disable the positivity checking of inductive types and the productivity checking of coinductive types. Warning: this can break the consistency of the system, use at your own risk. Unchecked (co)inductive types are printed by *Print Assumptions*.

Flag: `Universe Checking`

This flag can be used to enable/disable the checking of universes, providing a form of ”type in type”. Warning: this breaks the consistency of the system, use at your own risk. Constants relying on ”type in type” are printed by *Print Assumptions*. It has the same effect as `-type-in-type` command line argument (see *By command line options*).

Command: `Print Typing Flags`

Print the status of the three typing flags: guard checking, positivity checking and universe checking.

Example

```
Unset Guard Checking.
Print Typing Flags.
    check_guarded: false
    check_positive: true
    check_universes: true

Fixpoint f (n : nat) : False
  := f n.
```

(continues on next page)

(continued from previous page)

```
    f is defined
    f is recursively defined (decreasing on 1st argument)

Fixpoint ackermann (m n : nat) {struct m} : nat :=
  match m with
  | 0 => S n
  | S m =>
    match n with
    | 0 => ackermann m 1
    | S n => ackermann m (ackermann (S m) n)
    end
  end.
    ackermann is defined
    ackermann is recursively defined (decreasing on 1st argument)

Print Assumptions ackermann.
    Axioms:
    ackermann is assumed to be guarded.
```

Note that the proper way to define the Ackermann function is to use an inner fixpoint:

```
Fixpoint ack m :=
  fix ackm n :=
  match m with
  | 0 => S n
  | S m' =>
    match n with
    | 0 => ack m' 1
    | S n' => ack m' (ackm n')
    end
  end.
    ack is defined
    ack is recursively defined (decreasing on 1st argument)
```

5.1.14 Internal registration commands

Due to their internal nature, the commands that are presented in this section are not for general use. They are meant to appear only in standard libraries and in support libraries of plug-ins.

Exposing constants to OCaml libraries

Command: `Register` *qualid*$_1$ `as` *qualid*$_2$

This command exposes the constant *qualid*$_1$ to OCaml libraries under the name *qualid*$_2$. This constant can then be dynamically located calling `Coqlib.lib_ref "`*qualid*$_2$`"`; i.e., there is no need to known where is the constant defined (file, module, library, etc.).

As a special case, when the first segment of *qualid*$_2$ is `kernel`, the constant is exposed to the kernel. For instance, the `Int63` module features the following declaration:

```
Register bool as kernel.ind_bool.
```

This makes the kernel aware of what is the type of boolean values. This information is used for instance to define the return type of the `#int63_eq` primitive.

See also:

Primitive Integers

Inlining hints for the fast reduction machines

Command: `Register Inline` *qualid*

This command gives as a hint to the reduction machines (VM and native) that the body of the constant *qualid* should be inlined in the generated code.

Registering primitive operations

Command: `Primitive` $ident_1$ `:= #`$ident_2$`.`

Declares $ident_1$ as the primitive operator #$ident_2$. When running this command, the type of the primitive should be already known by the kernel (this is achieved through this command for primitive types and through the *Register* command with the `kernel` name-space for other types).

5.2 Proof handling

In Coq's proof editing mode all top-level commands documented in Chapter *Vernacular commands* remain available and the user has access to specialized commands dealing with proof development pragmas documented in this section. They 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 *Tactics*.

Coq user interfaces usually have a way of marking whether the user has switched to proof editing mode. For instance, in coqtop 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 called the *local context* of the goal. Initially, the local context contains the local variables and hypotheses of the current section (see Section *Assumptions*) and the local variables and hypotheses of the theorem statement. It is enriched by the use of certain tactics (see e.g. *intro*).

When a proof is completed, the message `Proof completed` is displayed. One can then register 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* [How80][Bar81][GLT89][Hue89], Coq stores proofs as terms of Cic. Those terms are called *proof terms*.

Error: `No focused proof.`

Coq raises this error message when one attempts to use a proof editing command out of the proof editing mode.

5.2.1 Switching on/off the proof editing mode

The proof editing mode is entered by asserting a statement, which typically is the assertion of a theorem using an assertion command like *Theorem*. The list of assertion commands is given in *Assertions and proofs*. The command *Goal* can also be used.

Command: `Goal` *`form`*

This is intended for quick assertion of statements, without knowing in advance which name to give to the assertion, typically for quick testing of the provability of a statement. If the proof of the statement is eventually completed and validated, the statement is then bound to the name `Unnamed_thm` (or a variant of this name not already used for another statement).

Command: `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: `Attempt to save an incomplete proof.`

Note: 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.

Variant: `Defined`

Same as *`Qed`* but the proof is then declared transparent, which means that its content can be explicitly used for type checking and that it can be unfolded in conversion tactics (see *Performing computations*, *`Opaque`*, *`Transparent`*).

Variant: `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.

Command: `Admitted`

This command is available in interactive editing mode to give up the current proof and declare the initial goal as an axiom.

Command: `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: `No focused proof (No proof-editing in progress).`

Variant: `Abort` *`ident`*

Aborts the editing of the proof named *`ident`* (in case you have nested proofs).

See also:

`Nested Proofs Allowed`

Variant: `Abort All`

Aborts all current goals.

Command: `Proof` *`term`*

This command applies in proof editing mode. It is equivalent to `exact` *`term`*`. Qed.` That is, you have to give the full proof in one gulp, as a proof term (see Section *Applying theorems*).

Command: `Proof`

Is a no-op which is useful to delimit the sequence of tactic commands which start a proof, after a *`Theorem`* command. It is a good practice to use *`Proof`* as an opening parenthesis, closed in the script with a closing *`Qed`*.

See also:

Proof with

Command: **`Proof using ident+`**

This command applies in proof editing mode. It declares the set of section variables (see *Assumptions*) used by the proof. At `Qed` time, the system will assert that the set of section variables actually used in the proof is a subset of the declared one.

The set of declared variables is closed under type dependency. For example, if `T` is a variable and `a` is a variable of type `T`, then the commands `Proof using a` and `Proof using T a` are equivalent.

Variant: **`Proof using ident+ with tactic`**

Combines in a single line `Proof with` and `Proof using`.

See also:

Setting implicit automation tactics

Variant: **`Proof using All`**

Use all section variables.

Variant: **`Proof using Type?`**

Use only section variables occurring in the statement.

Variant: **`Proof using Type*`**

The `*` operator computes the forward transitive closure. E.g. if the variable `H` has type `p < 5` then `H` is in `p*` since `p` occurs in the type of `H`. `Type*` is the forward transitive closure of the entire set of section variables occurring in the statement.

Variant: **`Proof using -(ident+)`**

Use all section variables except the list of `ident`.

Variant: **`Proof using collection₁ + collection₂`**

Use section variables from the union of both collections. See *Name a set of section hypotheses for Proof using* to know how to form a named collection.

Variant: **`Proof using collection₁ - collection₂`**

Use section variables which are in the first collection but not in the second one.

Variant: **`Proof using collection - (ident+)`**

Use section variables which are in the first collection but not in the list of `ident`.

Variant: **`Proof using collection *`**

Use section variables in the forward transitive closure of the collection. The `*` operator binds stronger than `+` and `-`.

Proof using options

The following options modify the behavior of `Proof using`.

Option: **`Default Proof Using "collection"`**

Use `collection` as the default `Proof using` value. E.g. `Set Default Proof Using "a b"` will complete all `Proof` commands not followed by a `using` part with `using a b`.

Flag: **`Suggest Proof Using`**

When `Qed` is performed, suggest a `using` annotation if the user did not provide one.

Name a set of section hypotheses for `Proof using`

Command: `Collection` *ident* `:=` *collection*

This can be used to name a set of section hypotheses, with the purpose of making `Proof using` annotations more compact.

Example

Define the collection named `Some` containing `x`, `y` and `z`:

```
Collection Some := x y z.
```

Define the collection named `Fewer` containing only `x` and `y`:

```
Collection Fewer := Some - z
```

Define the collection named `Many` containing the set union or set difference of `Fewer` and `Some`:

```
Collection Many := Fewer + Some
Collection Many := Fewer - Some
```

Define the collection named `Many` containing the set difference of `Fewer` and the unnamed collection `x y`:

```
Collection Many := Fewer - (x y)
```

Command: `Existential` *num* `:=` *term*

This command instantiates an existential variable. *num* is an index in the list of uninstantiated existential variables displayed by *Show Existentials*.

This command is intended to be used to instantiate existential variables when the proof is completed but some uninstantiated existential variables remain. To instantiate existential variables during proof edition, you should use the tactic *instantiate*.

Command: `Grab Existential Variables`

This command can be run when a proof has no more goal to be solved but has remaining uninstantiated existential variables. It takes every uninstantiated existential variable and turns it into a goal.

Proof modes

When entering proof mode through commands such as *Goal* and *Proof*, Coq picks by default the $\mathrm{L}_{\mathrm{tac}}$ mode. Nonetheless, there exist other proof modes shipped in the standard Coq installation, and furthermore some plugins define their own proof modes. The default proof mode used when opening a proof can be changed using the following option.

Option: `Default Proof Mode` *string*

Select the proof mode to use when starting a proof. Depending on the proof mode, various syntactic constructs are allowed when writing an interactive proof. The possible option values are listed below.

- ”Classic”: this is the default. It activates the $\mathrm{L}_{\mathrm{tac}}$ language to interact with the proof, and also allows vernacular commands.
- ”Noedit”: this proof mode only allows vernacular commands. No tactic language is activated at all. This is the default when the prelude is not loaded, e.g. through the `-noinit` option for `coqc`.
- ”Ltac2”: this proof mode is made available when requiring the Ltac2 library, and is set to be the default when it is imported. It allows to use the Ltac2 language, as well as vernacular commands.

- Some external plugins also define their own proof mode, which can be activated via this command.

5.2.2 Navigation in the proof tree

Command: `Undo`

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

Variant: `Undo` *`num`*

Repeats Undo *`num`* times.

Variant: `Restart`

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

Error: `No focused proof to restart.`

Command: `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.

Deprecated since version 8.8: Prefer the use of bullets or focusing brackets (see below).

Variant: `Focus` *`num`*

This focuses the attention on the *`num`* th subgoal to prove.

Deprecated since version 8.8: Prefer the use of focusing brackets with a goal selector (see below).

Command: `Unfocus`

This command restores to focus the goal that were suspended by the last *`Focus`* command.

Deprecated since version 8.8.

Command: `Unfocused`

Succeeds if the proof is fully unfocused, fails if there are some goals out of focus.

Command: `{ | }`

The command `{` (without a terminating period) focuses on the first goal, much like *`Focus`* does, however, the subproof can only be unfocused when it has been fully solved (*i.e.* when there is no focused goal left). Unfocusing is then handled by `}` (again, without a terminating period). See also an example in the next section.

Note that when a focused goal is proved a message is displayed together with a suggestion about the right bullet or `}` to unfocus it or focus the next one.

Variant: *`num`*`: {`

This focuses on the *`num`*-th subgoal to prove.

Variant: `[`*`ident`*`]: {`

This focuses on the named goal *`ident`*.

Note: Goals are just existential variables and existential variables do not get a name by default. You can give a name to a goal by using `refine ?[`*`ident`*`]`. You may also wrap this in an Ltac-definition like:

```
Ltac name_goal name := refine ?[name].
```

See also:

Existential variables

Example

This first example uses the Ltac definition above, and the named goals only serve for documentation.

```
Goal forall n, n + 0 = n.
    1 subgoal

      ============================
      forall n : nat, n + 0 = n

Proof.
induction n; [ name_goal base | name_goal step ].
    2 subgoals

      ============================
      0 + 0 = 0

    subgoal 2 is:
     S n + 0 = S n

[base]: {
    1 subgoal

      ============================
      0 + 0 = 0

reflexivity.
    This subproof is complete, but there are some unfocused goals.
    Try unfocusing with "}".

    1 subgoal

    subgoal 1 is:
     S n + 0 = S n

}

[step]: {
    1 subgoal

      n : nat
      IHn : n + 0 = n
      ============================
      S n + 0 = S n

simpl.
    1 subgoal

      n : nat
      IHn : n + 0 = n
      ============================
      S (n + 0) = S n

f_equal.
    1 subgoal
```

(continues on next page)

(continued from previous page)

```
      n : nat
      IHn : n + 0 = n
      ============================
      n + 0 = n

assumption.
    No more subgoals.

}
Qed.
    No more subgoals.
```

This can also be a way of focusing on a shelved goal, for instance:

```
Goal exists n : nat, n = n.
    1 subgoal

      ============================
      exists n : nat, n = n

eexists ?[x].
    1 focused subgoal
    (shelved: 1)

      ============================
      ?x = ?x

reflexivity.
    All the remaining goals are on the shelf.

    1 subgoal

    subgoal 1 is:
     nat

[x]: exact 0.
    No more subgoals.

Qed.
```

Error: `This proof is focused, but cannot be unfocused this way.`

You are trying to use } but the current subproof has not been fully solved.

Error: `No such goal (`*num*`).`

Error: `No such goal (`*ident*`).`

Error: `Brackets do not support multi-goal selectors.`

Brackets are used to focus on a single goal given either by its position or by its name if it has one.

See also:

The error messages about bullets below.

Bullets

Alternatively to `{` and `}`, proofs can be structured with bullets. The use of a bullet `b` for the first time focuses on the first goal `g`, the same bullet cannot be used again until the proof of `g` is completed, then it is mandatory to focus the next goal with `b`. The consequence is that `g` and all goals present when `g` was focused are focused with the same bullet `b`. See the example below.

Different bullets can be used to nest levels. The scope of bullet does not go beyond enclosing `{` and `}`, so bullets can be reused as further nesting levels provided they are delimited by these. Bullets are made of repeated `-`, `+` or `*` symbols:

```
bullet ::= -+ | ++ | *+
```

Note again that when a focused goal is proved a message is displayed together with a suggestion about the right bullet or `}` to unfocus it or focus the next one.

Note: In Proof General (`Emacs` interface to Coq), you must use bullets with the priority ordering shown above to have a correct indentation. For example `-` must be the outer bullet and `**` the inner one in the example below.

The following example script illustrates all these features:

Example

```
Goal (((True /\ True) /\ True) /\ True) /\ True.
    1 subgoal

      ============================
      (((True /\ True) /\ True) /\ True) /\ True

Proof.
split.
    2 subgoals

      ============================
      ((True /\ True) /\ True) /\ True

    subgoal 2 is:
     True

- split.
    1 subgoal

      ============================
      ((True /\ True) /\ True) /\ True

    2 subgoals

      ============================
      (True /\ True) /\ True

    subgoal 2 is:
     True

+ split.
```

(continues on next page)

(continued from previous page)

```
    1 subgoal

      ============================
      (True /\ True) /\ True

    2 subgoals

      ============================
      True /\ True

    subgoal 2 is:
     True

** { split.
    1 subgoal

      ============================
      True /\ True

    1 subgoal

      ============================
      True /\ True

    2 subgoals

      ============================
      True

    subgoal 2 is:
     True

- trivial.
    1 subgoal

      ============================
      True

    This subproof is complete, but there are some unfocused goals.
    Focus next goal with bullet -.

    4 subgoals

    subgoal 1 is:
     True
    subgoal 2 is:
     True
    subgoal 3 is:
     True
    subgoal 4 is:
     True

- trivial.
    1 subgoal

      ============================
```

(continues on next page)

(continued from previous page)

```
      True

    This subproof is complete, but there are some unfocused goals.
    Try unfocusing with "}".

    3 subgoals

    subgoal 1 is:
     True
    subgoal 2 is:
     True
    subgoal 3 is:
     True

}
** trivial.
    This subproof is complete, but there are some unfocused goals.
    Focus next goal with bullet **.

    3 subgoals

    subgoal 1 is:
     True
    subgoal 2 is:
     True
    subgoal 3 is:
     True

    1 subgoal

      ============================
      True

    This subproof is complete, but there are some unfocused goals.
    Focus next goal with bullet +.

    2 subgoals

    subgoal 1 is:
     True
    subgoal 2 is:
     True

+ trivial.
    1 subgoal

      ============================
      True

    This subproof is complete, but there are some unfocused goals.
    Focus next goal with bullet -.

    1 subgoal

    subgoal 1 is:
     True
```

(continues on next page)

(continued from previous page)

```
- assert True.
    1 subgoal

      ============================
      True

    2 subgoals

      ============================
      True

    subgoal 2 is:
     True

{ trivial.
    1 subgoal

      ============================
      True

    This subproof is complete, but there are some unfocused goals.
    Try unfocusing with "}".

    1 subgoal

    subgoal 1 is:
     True

}
assumption.
    1 subgoal

      H : True
      ============================
      True

    No more subgoals.

Qed.
```

Error: Wrong bullet *bullet*$_1$: Current bullet *bullet*$_2$ is not finished.
Before using bullet *bullet*$_1$ again, you should first finish proving the current focused goal. Note that *bullet*$_1$ and *bullet*$_2$ may be the same.

Error: Wrong bullet *bullet*$_1$: Bullet *bullet*$_2$ is mandatory here.
You must put *bullet*$_2$ to focus on the next goal. No other bullet is allowed here.

Error: No such goal. Focus next goal with bullet *bullet*.
You tried to apply a tactic but no goals were under focus. Using *bullet* is mandatory here.

Error: No such goal. Try unfocusing with }.
You just finished a goal focused by `{`, you must unfocus it with `}`.

Mandatory Bullets

Using *Default Goal Selector* with the ! selector forces tactic scripts to keep focus to exactly one goal (e.g. using bullets) or use explicit goal selectors.

Set Bullet Behavior

Option: `Bullet Behavior` `"None" | "Strict Subproofs"`

This option controls the bullet behavior and can take two possible values:

- ”None”: this makes bullets inactive.
- ”Strict Subproofs”: this makes bullets active (this is the default behavior).

5.2.3 Requesting information

Command: `Show`

This command displays the current goals.

Error: `No focused proof.`

Variant: `Show` *num*

Displays only the *num*-th subgoal.

Error: `No such goal.`

Variant: `Show` *ident*

Displays the named goal *ident*. This is useful in particular to display a shelved goal but only works if the corresponding existential variable has been named by the user (see *Existential variables*) as in the following example.

Example

```
Goal exists n, n = 0.
    1 subgoal

      ============================
      exists n : nat, n = 0

eexists ?[n].
    1 focused subgoal
    (shelved: 1)

      ============================
      ?n = 0

Show n.
    subgoal n is:

      ============================
      nat
```

Variant: `Show Proof` `[Diffs [removed]?]?`

Displays the proof term generated by the tactics that have been applied so far. If the proof is incomplete, the term will contain holes, which correspond to subterms which are still to be constructed. Each hole is an existential variable, which appears as a question mark followed by an identifier.

Experimental: Specifying "Diffs" highlights the difference between the current and previous proof step. By default, the command shows the output once with additions highlighted. Including "removed" shows the output twice: once showing removals and once showing additions. It does not examine the *Diffs* option. See *Showing differences between proof steps*.

Variant: `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.

Variant: `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.

Variant: `Show Intros`

This command is similar to the previous one, it simulates the naming process of an *intros*.

Variant: `Show Existentials`

Displays all open goals / existential variables in the current proof along with the type and the context of each variable.

Variant: `Show Match` *ident*

This variant displays a template of the Gallina `match` construct with a branch for each constructor of the type *ident*

Example

```
Show Match nat.
    match # with
     | O =>
     | S x =>
     end
```

Error: `Unknown inductive type.`

Variant: `Show Universes`

It displays the set of all universe constraints and its normalized form at the current stage of the proof, useful for debugging universe inconsistencies.

Variant: `Show Goal` *num* `at` *num*

This command is only available in coqtop. Displays a goal at a proof state using the goal ID number and the proof state ID number. It is primarily for use by tools such as Prooftree that need to fetch goal history in this way. Prooftree is a tool for visualizing a proof as a tree that runs in Proof General.

Command: `Guarded`

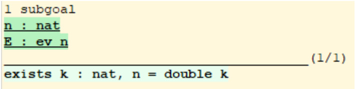
Some tactics (e.g. *refine*) allow to build proofs using fixpoint or co-fixpoint constructions. Due to the incremental nature of interactive proof construction, the check of the termination (or guardedness) of the recursive calls in the fixpoint or cofixpoint constructions is postponed to the time of the completion of the proof.

The command *Guarded* allows checking if the guard condition for fixpoint and cofixpoint is violated at some time of the construction of the proof without having to wait the completion of the proof.

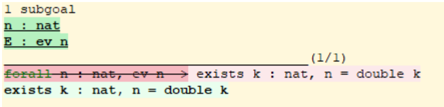
5.2.4 Showing differences between proof steps

Coq can automatically highlight the differences between successive proof steps and between values in some error messages. Also, as an experimental feature, Coq can also highlight differences between proof steps shown in the *Show Proof* command, but only, for now, when using coqtop and Proof General.

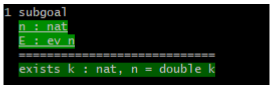
For example, the following screenshots of CoqIDE and coqtop show the application of the same *intros* tactic. The tactic creates two new hypotheses, highlighted in green. The conclusion is entirely in pale green because although it's changed, no tokens were added to it. The second screenshot uses the "removed" option, so it shows the conclusion a second time with the old text, with deletions marked in red. Also, since the hypotheses are new, no line of old text is shown for them.



```
1 subgoal
n : nat
E : ev n
______________________________________(1/1)
exists k : nat, n = double k
```

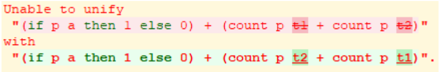


```
1 subgoal
n : nat
E : ev n
______________________________________(1/1)
~~forall n : nat, ev n ->~~ exists k : nat, n = double k
exists k : nat, n = double k
```



```
1 subgoal
  n : nat
  E : ev n
  ============================
  exists k : nat, n = double k
```

This image shows an error message with diff highlighting in CoqIDE:



```
Unable to unify
 "(if p a then 1 else 0) + (count p ~~t1~~ + count p ~~t2~~)"
with
 "(if p a then 1 else 0) + (count p t2 + count p t1)".
```

How to enable diffs

Option: `Diffs` "on" | "off" | "removed"

The "on" setting highlights added tokens in green, while the "removed" setting additionally reprints items with removed tokens in red. Unchanged tokens in modified items are shown with pale green or red. Diffs in error messages use red and green for the compared values; they appear regardless of the setting. (Colors are user-configurable.)

For coqtop, showing diffs can be enabled when starting coqtop with the `-diffs on|off|removed` command-line option or by setting the *Diffs* option within Coq. You will need to provide the `-color on|auto` command-line option when you start coqtop in either case.

Colors for coqtop can be configured by setting the `COQ_COLORS` environment variable. See section *By environment variables*. Diffs use the tags `diff.added`, `diff.added.bg`, `diff.removed` and `diff.removed.bg`.

In CoqIDE, diffs should be enabled from the `View` menu. Don't use the `Set Diffs` command in CoqIDE. You can change the background colors shown for diffs from the `Edit | Preferences | Tags` panel by changing the settings for the `diff.added`, `diff.added.bg`, `diff.removed` and `diff.removed.bg` tags. This panel also lets you control other attributes of the highlights, such as the foreground color, bold, italic, underline and strikeout.

As of June 2019, Proof General can also display Coq-generated proof diffs automatically. Please see the PG documentation section "Showing Proof Diffs"[295]) for details.

[295] https://proofgeneral.github.io/doc/master/userman/Coq-Proof-General#Showing-Proof-Diffs

How diffs are calculated

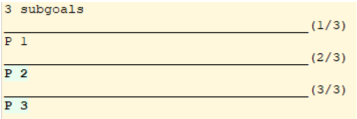
Diffs are calculated as follows:

1. Select the old proof state to compare to, which is the proof state before the last tactic that changed the proof. Changes that only affect the view of the proof, such as `all: swap 1 2`, are ignored.
2. For each goal in the new proof state, determine what old goal to compare it to—the one it is derived from or is the same as. Match the hypotheses by name (order is ignored), handling compacted items specially.
3. For each hypothesis and conclusion (the "items") in each goal, pass them as strings to the lexer to break them into tokens. Then apply the Myers diff algorithm *[Mye86]* on the tokens and add appropriate highlighting.

Notes:

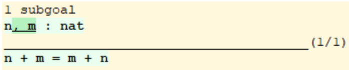
- Aside from the highlights, output for the "on" option should be identical to the undiffed output.
- Goals completed in the last proof step will not be shown even with the "removed" setting.

This screen shot shows the result of applying a *split* tactic that replaces one goal with 2 goals. Notice that the goal `P 1` is not highlighted at all after the split because it has not changed.



```
3 subgoals
______________________________________(1/3)
P 1
______________________________________(2/3)
P 2
______________________________________(3/3)
P 3
```

This is how diffs may appear after applying a *intro* tactic that results in compacted hypotheses:



```
1 subgoal
n, m : nat
______________________________________(1/1)
n + m = m + n
```

5.2.5 Controlling the effect of proof editing commands

Option: `Hyps Limit` *`num`*

This option controls the maximum number of hypotheses displayed in goals after the application of a tactic. All the hypotheses remain usable in the proof development. When unset, it goes back to the default mode which is to print all available hypotheses.

Flag: `Nested Proofs Allowed`

When turned on (it is off by default), this flag enables support for nested proofs: a new assertion command can be inserted before the current proof is finished, in which case Coq will temporarily switch to the proof of this *nested lemma*. When the proof of the nested lemma is finished (with *`Qed`* or *`Defined`*), its statement will be made available (as if it had been proved before starting the previous proof) and Coq will switch back to the proof of the previous assertion.

5.2.6 Controlling memory usage

When experiencing high memory usage the following commands can be used to force Coq to optimize some of its internal data structures.

Command: `Optimize Proof`

This command forces Coq to shrink the data structure used to represent the ongoing proof.

Command: `Optimize Heap`

This command forces the OCaml runtime to perform a heap compaction. This is in general an expensive operation. See: OCaml Gc[296] There is also an analogous tactic *optimize_heap*.

5.3 Tactics

A deduction rule is a link between some (unique) formula, that we call the *conclusion* and (several) formulas that we call the *premises*. A deduction rule can be read in two ways. The first one says: "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 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 B, I have to prove A and I have to prove B. This is backward reasoning 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 *Requesting information*).

Since not every rule applies to a given statement, not every tactic can be used to reduce a given 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 built 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 *Ltac*.

5.3.1 Common elements of tactics

Invocation of tactics

A tactic is applied as an ordinary command. It may be preceded by a goal selector (see Section *Semantics*). If no selector is specified, the default selector is used.

```
tactic_invocation  ::=  toplevel_selector : tactic.
                        tactic.
```

Option: `Default Goal Selector "toplevel_selector"`

This option controls the default selector, used when no selector is specified when applying a tactic. The initial value is 1, hence the tactics are, by default, applied to the first goal.

Using value `all` will make it so that tactics are, by default, applied to every goal simultaneously. Then, to apply a tactic tac to the first goal only, you can write `1:tac`.

Using value `!` enforces that all tactics are used either on a single focused goal or with a local selector (''strict focusing mode'').

Although more selectors are available, only `all`, `!` or a single natural number are valid default goal selectors.

[296] http://caml.inria.fr/pub/docs/manual-ocaml/libref/Gc.html#VALcompact

Bindings list

Tactics that take a term as an argument may also support a bindings list to instantiate some parameters of the term by name or position. The general form of a term with a bindings list is *term* `with` *bindings_list* where *bindings_list* can take two different forms:

```
ref             ::=   ident
                      num
bindings_list   ::=   (ref := term) ... (ref := term)
                      term ... term
```

- In a bindings list of the form `(ref:= term)`+ , *ref* is either an *ident* or a *num*. The references are determined according to the type of *term*. If *ref* is an identifier, this identifier has to be bound in the type of *term* and the binding provides the tactic with an instance for the parameter of this name. If *ref* is a number `n`, it refers to the `n`-th non dependent premise of the *term*, as determined by the type of *term*.

  Error: `No such binder.`

- A bindings list can also be a simple list of terms `term`*. In that case the references to which these terms correspond are determined by the tactic. In case of *induction*, *destruct*, *elim* and *case*, the terms have to provide instances for all the dependent products in the type of term while in the case of *apply*, or of *constructor* and its variants, only instances for the dependent products that are not bound in the conclusion of the type are required.

  Error: `Not the right number of missing arguments.`

Intro patterns

Intro patterns let you specify the name to assign to variables and hypotheses introduced by tactics. They also let you split an introduced hypothesis into multiple hypotheses or subgoals. Common tactics that accept intro patterns include *assert*, *intros* and *destruct*.

```
intropattern_list            ::=   intropattern ... intropattern
                                   empty
empty                        ::=
intropattern                 ::=   *
                                   **
                                   simple_intropattern
simple_intropattern          ::=   simple_intropattern_closed [ % term ... % term ]
simple_intropattern_closed   ::=   naming_intropattern
                                   _
                                   or_and_intropattern
                                   rewriting_intropattern
                                   injection_intropattern
naming_intropattern          ::=   ident
                                   ?
                                   ?ident
or_and_intropattern          ::=   [ intropattern_list | ... | intropattern_list ]
                                   ( simple_intropattern , ... , simple_intropattern )
                                   ( simple_intropattern & ... & simple_intropattern )
rewriting_intropattern       ::=   ->
```

```
                                  <-
injection_intropattern      ::=   [= intropattern_list ]
or_and_intropattern_loc     ::=   or_and_intropattern
                                  ident
```

Note that the intro pattern syntax varies between tactics. Most tactics use *simple_intropattern* in the grammar. *destruct*, *edestruct*, *induction*, *einduction*, *case*, *ecase* and the various *inversion* tactics use *or_and_intropattern_loc*, while *intros* and *eintros* use *intropattern_list*. The `eqn:` construct in various tactics uses *naming_intropattern*.

Naming patterns

Use these elementary patterns to specify a name:

- *ident* — use the specified name
- `?` — let Coq choose a name
- `?`*ident* — generate a name that begins with *ident*
- `_` — discard the matched part (unless it is required for another hypothesis)
- if a disjunction pattern omits a name, such as `[|H2]`, Coq will choose a name

Splitting patterns

The most common splitting patterns are:

- split a hypothesis in the form `A /\ B` into two hypotheses `H1: A` and `H2: B` using the pattern `(H1 & H2)` or `(H1, H2)` or `[H1 H2]`. *Example*. This also works on `A <-> B`, which is just a notation representing `(A -> B) /\ (B -> A)`.
- split a hypothesis in the form `A \/ B` into two subgoals using the pattern `[H1|H2]`. The first subgoal will have the hypothesis `H1: A` and the second subgoal will have the hypothesis `H2: B`. *Example*
- split a hypothesis in either of the forms `A /\ B` or `A \/ B` using the pattern `[]`.

Patterns can be nested: `[[Ha|Hb] H]` can be used to split `(A \/ B) /\ C`.

Note that there is no equivalent to intro patterns for goals. For a goal `A /\ B`, use the *split* tactic to replace the current goal with subgoals `A` and `B`. For a goal `A \/ B`, use *left* to replace the current goal with `A`, or *right* to replace the current goal with `B`.

- `(` *simple_intropattern*+`,` `)` — matches a product over an inductive type with a *single constructor*. If the number of patterns equals the number of constructor arguments, then it applies the patterns only to the arguments, and `(` *simple_intropattern*+`,` `)` is equivalent to `[`*simple_intropattern*+`]`. If the number of patterns equals the number of constructor arguments plus the number of `let-ins`, the patterns are applied to the arguments and `let-in` variables.
- `(` *simple_intropattern*+`&` `)` — matches a right-hand nested term that consists of one or more nested binary inductive types such as `a1 OP1 a2 OP2 ...` (where the `OPn` are right-associative). (If the `OPn` are left-associative, additional parentheses will be needed to make the term right-hand nested, such as `a1 OP1 (a2 OP2 ...)`.) The splitting pattern can have more than 2 names, for example `(H1 & H2 & H3)` matches `A /\ B /\ C`. The inductive types must have a *single constructor with two parameters*. *Example*
- `[` *intropattern_list*+`|` `]` — splits an inductive type that has *multiple constructors* such as `A \/ B` into multiple subgoals. The number of *intropattern_list* must be the same as the number of constructors for the matched part.

- `[` *intropattern*`+` `]` — splits an inductive type that has a *single constructor with multiple parameters* such as `A /\ B` into multiple hypotheses. Use `[H1 [H2 H3]]` to match `A /\ B /\ C`.
- `[]` — splits an inductive type: If the inductive type has multiple constructors, such as `A \/ B`, create one subgoal for each constructor. If the inductive type has a single constructor with multiple parameters, such as `A /\ B`, split it into multiple hypotheses.

Equality patterns

These patterns can be used when the hypothesis is an equality:

- `->` — replaces the right-hand side of the hypothesis with the left-hand side of the hypothesis in the conclusion of the goal; the hypothesis is cleared; if the left-hand side of the hypothesis is a variable, it is substituted everywhere in the context and the variable is removed. *Example*
- `<-` — similar to `->`, but replaces the left-hand side of the hypothesis with the right-hand side of the hypothesis.
- `[=` *intropattern*`*,` `]` — If the product is over an equality type, applies either *injection* or *discriminate*. If *injection* is applicable, the intropattern is used on the hypotheses generated by *injection*. If the number of patterns is smaller than the number of hypotheses generated, the pattern `?` is used to complete the list. *Example*

Other patterns

- `*` — introduces one or more quantified variables from the result until there are no more quantified variables. *Example*
- `**` — introduces one or more quantified variables or hypotheses from the result until there are no more quantified variables or implications (`->`). `intros **` is equivalent to `intros`. *Example*
- *simple_intropattern_closed* `% term*` — first applies each of the terms with the *apply ... in* tactic on the hypothesis to be introduced, then it uses *simple_intropattern_closed*. *Example*

Flag: `Bracketing Last Introduction Pattern`

For `intros` *intropattern_list*, controls how to handle a conjunctive pattern that doesn't give enough simple patterns to match all the arguments in the constructor. If set (the default), Coq generates additional names to match the number of arguments. Unsetting the flag will put the additional hypotheses in the goal instead, behavior that is more similar to SSReflect's intro patterns.

Deprecated since version 8.10.

Note: `A \/ B` and `A /\ B` use infix notation to refer to the inductive types `or` and `and`. `or` has multiple constructors (`or_introl` and `or_intror`), while `and` has a single constructor (`conj`) with multiple parameters (`A` and `B`). These are defined in theories/Init/Logic.v. The ”where” clauses define the infix notation for ”or” and ”and”.

```
Inductive or (A B:Prop) : Prop :=
  | or_introl : A -> A \/ B
  | or_intror : B -> A \/ B
where "A \/ B" := (or A B) : type_scope.

Inductive and (A B:Prop) : Prop :=
  conj : A -> B -> A /\ B
where "A /\ B" := (and A B) : type_scope.
```

Note: `intros` `p`$^+$ is not always equivalent to `intros p; ... ; intros p` if some of the `p` are `_`. In the first form, all erasures are done at once, while they're done sequentially for each tactic in the second form. If the second matched term depends on the first matched term and the pattern for both is `_` (i.e., both will be erased), the first `intros` in the second form will fail because the second matched term still has the dependency on the first.

Examples:

Example: intro pattern for /

```
    1 subgoal

      A, B : Prop
      H : A /\ B
      ============================
      True

destruct H as (HA & HB).
    1 subgoal

      A, B : Prop
      HA : A
      HB : B
      ============================
      True
```

Example: intro pattern for \/

```
    1 subgoal

      A, B : Prop
      H : A \/ B
      ============================
      True

destruct H as [HA|HB].
    2 subgoals

      A, B : Prop
      HA : A
      ============================
      True

    subgoal 2 is:
     True

all: swap 1 2.
    2 subgoals

      A, B : Prop
      HB : B
      ============================
```

(continues on next page)

(continued from previous page)

```
      True

    subgoal 2 is:
     True
```

Example: -> intro pattern

```
    1 subgoal

      x, y, z : nat
      H : x = y
      ============================
      y = z -> x = z

intros ->.
    1 subgoal

      x, z : nat
      H : x = z
      ============================
      x = z
```

Example: [=] intro pattern

The first `intros [=]` uses *`injection`* to strip `(S ...)` from both sides of the matched equality. The second uses *`discriminate`* on the contradiction `1 = 2` (internally represented as `(S O) = (S (S O))`) to complete the goal.

```
    1 subgoal

      n, m : nat
      ============================
      S n = S m -> 1 = 2 -> False

intros [= H].
    1 subgoal

      n, m : nat
      H : n = m
      ============================
      1 = 2 -> False

intros [=].
    No more subgoals.
```

Example: (A & B & …) intro pattern

```
    1 subgoal
```

(continues on next page)

(continued from previous page)

```
      ============================
      A /\ (exists x : nat, B x /\ C) -> True

intros (a & x & b & c).
    1 subgoal

      a : A
      x : nat
      b : B x
      c : C
      ============================
      True
```

Example: * intro pattern

```
    1 subgoal

      ============================
      forall A B : Prop, A -> B

intros *.
    1 subgoal

      A, B : Prop
      ============================
      A -> B
```

Example: ** pattern ("intros **" is equivalent to "intros")

```
    1 subgoal

      ============================
      forall A B : Prop, A -> B

intros **.
    1 subgoal

      A, B : Prop
      H : A
      ============================
      B
```

Example: compound intro pattern

```
    1 subgoal

      ============================
      forall A B C : Prop, A \/ B /\ C -> (A -> C) -> C
```

```
intros * [a | (_,c)] f.
    2 subgoals

      A, B, C : Prop
      a : A
      f : A -> C
      ============================
      C

    subgoal 2 is:
     C

all: swap 1 2.
    2 subgoals

      A, B, C : Prop
      c : C
      f : A -> C
      ============================
      C

    subgoal 2 is:
     C
```

Example: combined intro pattern using [=] -> and %

```
    1 subgoal

      A : Type
      xs, ys : list A
      ============================
      S (length ys) = 1 -> xs ++ ys = xs

intros [=->%length_zero_iff_nil].
    1 subgoal

      A : Type
      xs : list A
      ============================
      xs ++ nil = xs
```

- `intros` would add `H : S (length ys) = 1`
- `intros [=]` would additionally apply *`injection`* to `H` to yield `H0 : length ys = 0`
- `intros [=->%length_zero_iff_nil]` applies the theorem, making H the equality `l=nil`, which is then applied as for `->`.

```
Theorem length_zero_iff_nil (l : list A):
   length l = 0 <-> l=nil.
```

The example is based on Tej Chajed's coq-tricks[297]

[297] https://github.com/tchajed/coq-tricks/blob/8e6efe4971ed828ac8bdb5512c1f615d7d62691e/src/IntroPatterns.v

Occurrence sets and occurrence clauses

An occurrence clause is a modifier to some tactics that obeys the following syntax:

```
occurrence_clause   ::=   in goal_occurrences
goal_occurrences    ::=   [ident [at_occurrences], ... , ident [at_occurrences] [|- [* [at_occurrences]
                          * |- [* [at_occurrences]]
                          *
at_occurrences      ::=   at occurrences
occurrences         ::=   [-] num ... num
```

The role of an occurrence clause is to select a set of occurrences of a term in a goal. In the first case, the *ident* at num * ? parts indicate that occurrences have to be selected in the hypotheses named *ident*. If no numbers are given for hypothesis *ident*, then all the occurrences of *term* in the hypothesis are selected. If numbers are given, they refer to occurrences of *term* when the term is printed using the *Printing All* flag, counting from left to right. In particular, occurrences of *term* in implicit arguments (see *Implicit arguments*) or coercions (see *Coercions*) are counted.

If a minus sign is given between `at` and the list of occurrences, it negates the condition so that the clause denotes all the occurrences except the ones explicitly mentioned after the minus sign.

As an exception to the left-to-right order, the occurrences in the return subexpression of a match are considered *before* the occurrences in the matched term.

In the second case, the `*` on the left of `|-` means that all occurrences of term are selected in every hypothesis.

In the first and second case, if `*` is mentioned on the right of `|-`, the occurrences of the conclusion of the goal have to be selected. If some numbers are given, then only the occurrences denoted by these numbers are selected. If no numbers are given, all occurrences of *term* in the goal are selected.

Finally, the last notation is an abbreviation for `* |- *`. Note also that `|-` is optional in the first case when no `*` is given.

Here are some tactics that understand occurrence clauses: *set*, *remember*, *induction*, *destruct*.

See also:

Managing the local context, *Case analysis and induction*, *Printing constructions in full*.

5.3.2 Applying theorems

`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 *Conversion rules*).

Error: `Not an exact proof.`

Variant: `eexact` *term*.

This tactic behaves like *exact* but is able to handle terms and goals with existential variables.

`assumption`

This tactic looks in the local context for a hypothesis whose type is convertible to the goal. If it is the case, the subgoal is proved. Otherwise, it fails.

Error: `No such assumption.`

Variant: eassumption

This tactic behaves like *assumption* but is able to handle goals with existential variables.

refine *term*

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 there are remaining holes in the elaborated term. The type of holes must be either synthesized by the system or declared by an explicit cast like (_ : nat -> Prop). Any subgoal that occurs in other subgoals is automatically shelved, as if calling *shelve_unifiable*. The produced subgoals (shelved or not) are *not* candidates for typeclass resolution, even if they have a type-class type as conclusion, letting the user control when and how typeclass resolution is launched on them. This low-level tactic can be useful to advanced users.

Example

```
Inductive Option : Set :=
| Fail : Option
| Ok : bool -> Option.
    Option is defined
    Option_rect is defined
    Option_ind is defined
    Option_rec is defined
    Option_sind is defined

Definition get : forall x:Option, x <> Fail -> bool.
    1 subgoal

      ============================
      forall x : Option, x <> Fail -> bool

refine
    (fun x:Option =>
      match x return x <> Fail -> bool with
      | Fail => _
      | Ok b => fun _ => b
      end).
    1 subgoal

      x : Option
      ============================
      Fail <> Fail -> bool

intros; absurd (Fail = Fail); trivial.
    No more subgoals.

Defined.
```

Error: Invalid argument.

The tactic *refine* does not know what to do with the term you gave.

Error: 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 that call *refine* internally.

Error: Cannot infer a term for this placeholder.

There is a hole in the term you gave whose type cannot be inferred. Put a cast around it.

Variant: **simple refine** *term*

This tactic behaves like refine, but it does not shelve any subgoal. It does not perform any beta-reduction either.

Variant: **notypeclasses refine** *term*

This tactic behaves like *refine* except it performs type checking without resolution of typeclasses.

Variant: **simple notypeclasses refine** *term*

This tactic behaves like the combination of *simple refine* and *notypeclasses refine*: it performs type checking without resolution of typeclasses, does not perform beta reductions or shelve the subgoals.

Flag: **Debug Unification**

Enables printing traces of unification steps used during elaboration/typechecking and the *refine* tactic.

apply *term*

This tactic applies to any goal. The argument term is a term well-formed in the local context. The tactic *apply* tries to match the current goal against the conclusion of the type of *term*. If it succeeds, then the tactic returns as many subgoals as the number of non-dependent premises of the type of term. If the conclusion of the type of *term* does not match the goal *and* the conclusion is an inductive type isomorphic to a tuple type, then each component of the tuple is recursively matched to the goal in the left-to-right order.

The tactic *apply* relies on first-order unification with dependent types unless the conclusion of the type of *term* is of the form P (t$_1$ ... t$_n$) with P to be instantiated. In the latter case, the behavior depends on the form of the goal. If the goal is of the form (fun x => Q) u$_1$ ... u$_n$ and the t$_i$ and u$_i$ unify, then P is taken to be (fun x => Q). Otherwise, *apply* tries to define P by abstracting over t_1 ... t__n in the goal. See *pattern* to transform the goal so that it gets the form (fun x => Q) u$_1$ ... u$_n$.

Error: **Unable to unify** *term* **with** *term***.**

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.

Error: **Unable to find an instance for the variables** *ident*+ **.**

This occurs when some instantiations of the premises of *term* are not deducible from the unification. This is the case, for instance, when you want to apply a transitivity property. In this case, you have to use one of the variants below:

Variant: **apply** *term* **with** *term*+

Provides apply with explicit instantiations for all dependent premises of the type of term that do not occur in the conclusion and consequently cannot be found by unification. Notice that the collection *term*+ must be given according to the order of these dependent premises of the type of term.

Error: **Not the right number of missing arguments.**

Variant: **apply** *term* **with** *bindings_list*

This also provides apply with values for instantiating premises. Here, variables are referred by names and non-dependent products by increasing numbers (see *bindings list*).

Variant: **apply** *term*+,

This is a shortcut for apply *term*$_1$; [.. | ... ; [.. | apply *term*$_n$] ...], i.e. for the successive applications of *term*$_{i+1}$ on the last subgoal generated by apply *term*$_i$, starting from the application of *term*$_1$.

Variant: `eapply` *`term`*

The tactic *`eapply`* behaves like *`apply`* but it does not fail when no instantiations are deducible for some variables in the premises. Rather, it turns these variables into existential variables which are variables still to instantiate (see *Existential variables*). The instantiation is intended to be found later in the proof.

Variant: `simple apply` *`term`*.

This behaves like *`apply`* but it reasons modulo conversion only on subterms that contain no variables to instantiate. For instance, the following example does not succeed because it would require the conversion of `id ?foo` and `O`.

Example

```
Definition id (x : nat) := x.
    id is defined

Parameter H : forall y, id y = y.
    H is declared

Goal O = O.
    1 subgoal

      ============================
      O = O

Fail simple apply H.
    The command has indeed failed with message:
    Unable to unify "id ?M160 = ?M160" with "O = O".
```

Because it reasons modulo a limited amount of conversion, *`simple apply`* fails quicker than *`apply`* and it is then well-suited for uses in user-defined tactics that backtrack often. Moreover, it does not traverse tuples as *`apply`* does.

Variant: `simple`? `apply` *`term`* `with` *`bindings_list`*?+,

Variant: `simple`? `eapply` *`term`* `with` *`bindings_list`*?+,

This summarizes the different syntaxes for *`apply`* and *`eapply`*.

Variant: `lapply` *`term`*

This tactic applies to any goal, say `G`. The argument term has to be well-formed in the current context, its type being reducible to a non-dependent product `A -> B` with `B` possibly containing products. Then it generates two subgoals `B->G` and `A`. Applying `lapply H` (where `H` has type `A->B` and `B` does not start with a product) does the same as giving the sequence `cut B. 2:apply H.` where `cut` is described below.

Warning: `When` *`term`* `contains more than one non dependent product the tactic lapply only takes i`

Example

Assume we have a transitive relation `R` on `nat`:

```
Parameter R : nat -> nat -> Prop.
Axiom Rtrans : forall x y z:nat, R x y -> R y z -> R x z.
```

(continues on next page)

(continued from previous page)

```
Parameters n m p : nat.
Axiom Rnm : R n m.
Axiom Rmp : R m p.
```

Consider the goal `(R n p)` provable using the transitivity of `R`:

```
Goal R n p.
```

The direct application of `Rtrans` with `apply` fails because no value for `y` in `Rtrans` is found by `apply`:

```
apply Rtrans.
    Toplevel input, characters 0-12:
    > apply Rtrans.
    > ^^^^^^^^^^^^
    Error: Unable to find an instance for the variable y.
```

A solution is to `apply (Rtrans n m p)` or `(Rtrans n m)`.

```
apply (Rtrans n m p).
    2 subgoals

      ============================
      R n m

    subgoal 2 is:
     R m p
```

Note that `n` can be inferred from the goal, so the following would work too.

```
apply (Rtrans _ m).
```

More elegantly, `apply Rtrans with (y:=m)` allows only mentioning the unknown m:

```
apply Rtrans with (y := m).
```

Another solution is to mention the proof of `(R x y)` in `Rtrans`

```
apply Rtrans with (1 := Rnm).
    1 subgoal

      ============================
      R m p
```

... or the proof of `(R y z)`.

```
apply Rtrans with (2 := Rmp).
    1 subgoal

      ============================
      R n m
```

On the opposite, one can use `eapply` which postpones the problem of finding `m`. Then one can apply the hypotheses `Rnm` and `Rmp`. This instantiates the existential variable and completes the proof.

```
eapply Rtrans.
    2 focused subgoals
    (shelved: 1)
```

(continues on next page)

(continued from previous page)

```
      ============================
      R n ?y

    subgoal 2 is:
     R ?y p

apply Rnm.
    1 subgoal

      ============================
      R m p

apply Rmp.
    No more subgoals.
```

Note: When the conclusion of the type of the term to `apply` is an inductive type isomorphic to a tuple type and `apply` looks recursively whether a component of the tuple matches the goal, it excludes components whose statement would result in applying an universal lemma of the form `forall A, ... -> A`. Excluding this kind of lemma can be avoided by setting the following flag:

Flag: `Universal Lemma Under Conjunction`

This flag, which preserves compatibility with versions of Coq prior to 8.4 is also available for `apply` *`term`* `in` *`ident`* (see *`apply ... in`*).

`apply` *`term`* `in` *`ident`*

This tactic applies to any goal. The argument *`term`* is a term well-formed in the local context and the argument *`ident`* is an hypothesis of the context. The tactic `apply` *`term`* `in` *`ident`* tries to match the conclusion of the type of *`ident`* against a non-dependent premise of the type of *`term`*, trying them from right to left. If it succeeds, the statement of hypothesis *`ident`* is replaced by the conclusion of the type of *`term`*. The tactic also returns as many subgoals as the number of other non-dependent premises in the type of *`term`* and of the non-dependent premises of the type of *`ident`*. If the conclusion of the type of *`term`* does not match the goal *and* the conclusion is an inductive type isomorphic to a tuple type, then the tuple is (recursively) decomposed and the first component of the tuple of which a non-dependent premise matches the conclusion of the type of *`ident`*. Tuples are decomposed in a width-first left-to-right order (for instance if the type of `H1` is `A <-> B` and the type of `H2` is `A` then `apply H1 in H2` transforms the type of `H2` into `B`). The tactic *`apply`* relies on first-order pattern matching with dependent types.

Error: `Statement without assumptions.`

This happens if the type of *`term`* has no non-dependent premise.

Error: `Unable to apply.`

This happens if the conclusion of *`ident`* does not match any of the non-dependent premises of the type of *`term`*.

Variant: `apply` *`term`*+, `in` *`ident`*

This applies each *`term`* in sequence in *`ident`*.

Variant: `apply` *`term`* `with` *`bindings_list`*+, `in` *`ident`*

This does the same but uses the bindings in each (*`ident`* `:=` *`term`*) to instantiate the parameters of the corresponding type of *`term`* (see *bindings list*).

Variant: `eapply {+, term {? with bindings_list}} in ident`

This works as *apply … in* but turns unresolved bindings into existential variables, if any, instead of failing.

Variant: `apply {+, term {? with bindings_list}} in ident as simple_intropattern`

This works as *apply … in* then applies the *simple_intropattern* to the hypothesis *ident*.

Variant: `simple apply term in ident`

This behaves like *apply … in* but it reasons modulo conversion only on subterms that contain no variables to instantiate. For instance, if `id := fun x:nat => x` and `H: forall y, id y = y -> True` and `H0 : O = O` then `simple apply H in H0` does not succeed because it would require the conversion of `id ?x` and `O` where `?x` is an existential variable to instantiate. Tactic `simple apply` *term* `in` *ident* does not either traverse tuples as `apply` *term* `in` *ident* does.

Variant: `{? simple} apply {+, term {? with bindings_list}} in ident {? as simple_intropattern}`

Variant: `{? simple} eapply {+, term {? with bindings_list}} in ident {? as simple_intropattern}`

This summarizes the different syntactic variants of `apply` *term* `in` *ident* and `eapply` *term* `in` *ident*.

`constructor num`

This tactic applies to a goal such that its conclusion is an inductive type (say `I`). The argument *num* must be less or equal to the numbers of constructor(s) of `I`. Let `c`$_\mathtt{i}$ be the i-th constructor of `I`, then `constructor i` is equivalent to `intros; apply c`$_\mathtt{i}$.

Error: `Not an inductive product.`

Error: `Not enough constructors.`

Variant: `constructor`

This tries `constructor 1` then `constructor 2`, ..., then `constructor n` where `n` is the number of constructors of the head of the goal.

Variant: `constructor num with bindings_list`

Let `c` be the i-th constructor of `I`, then `constructor i with` *bindings_list* is equivalent to `intros; apply c 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).

Variant: `split {? with bindings_list}`

This applies only if `I` has a single constructor. It is then equivalent to `constructor 1 {? with bindings_list}`. It is typically used in the case of a conjunction $A \land B$.

Variant: `exists bindings_list`

This applies only if `I` has a single constructor. It is then equivalent to `intros; constructor 1 with` *bindings_list*. It is typically used in the case of an existential quantification $\exists x, P(x)$.

Variant: `exists` [*bindings_list*]$^{+}_{,}$

This iteratively applies `exists` *bindings_list*.

Error: `Not an inductive goal with 1 constructor.`

Variant: `left` [`with` *bindings_list*]$^{?}$

Variant: `right` [`with` *bindings_list*]$^{?}$

These tactics apply only if `I` has two constructors, for instance in the case of a disjunction $A \vee B$. Then, they are respectively equivalent to `constructor 1` [`with` *bindings_list*]$^{?}$ and `constructor 2` [`with` *bindings_list*]$^{?}$.

Error: `Not an inductive goal with 2 constructors.`

Variant: `econstructor`
Variant: `eexists`
Variant: `esplit`
Variant: `eleft`
Variant: `eright`

These tactics and their variants behave like *constructor*, *exists*, *split*, *left*, *right* and their variants but they introduce existential variables instead of failing when the instantiation of a variable cannot be found (cf. *eapply* and *apply*).

Flag: `Debug Tactic Unification`

Enables printing traces of unification steps in tactic unification. Tactic unification is used in tactics such as *apply* and *rewrite*.

5.3.3 Managing the local context

`intro`

This tactic applies to a goal that is either a product or starts with a let-binder. If the goal is a product, the tactic implements the "Lam" rule given in *Typing rules*[1]. 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 `Hn:T` (if `T` is of type `Set` or `Prop`) or `Xn:T` (if the type of `T` is `Type`). The optional index `n` is such that `Hn` or `Xn` is a fresh identifier. In both cases, the new subgoal is `U`.

If the goal is an existential variable, *intro* forces the resolution of the existential variable into a dependent product `∀ x:?X, ?Y`, puts `x:?X` in the local context and leaves `?Y` as a new subgoal allowed to depend on `x`.

The tactic *intro* applies the tactic *hnf* until *intro* can be applied or the goal is not head-reducible.

Error: `No product even after head-reduction.`

Variant: `intro` *ident*

This applies *intro* but forces *ident* to be the name of the introduced hypothesis.

Error: *ident* `is already used.`

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

Note: 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 *Qualified names*).

Variant: `intros`

This repeats *intro* until it meets the head-constant. It never reduces head-constants and it never fails.

Variant: `intros` *ident*$^+$ `.`

This is equivalent to the composed tactic `intro` *ident*`; ... ; intro` *ident*.

Variant: `intros until` *ident*

This repeats intro until it meets a premise of the goal having the form `(`*ident* `:` *type*`)` and discharges the variable named *ident* of the current goal.

Error: `No such hypothesis in current goal.`

Variant: `intros until` *num*

This repeats *intro* until the *num*-th non-dependent product.

Example

On the subgoal `forall x y : nat, x = y -> y = x` the tactic `intros until 1` is equivalent to `intros x y H`, as `x = y -> y = x` is the first non-dependent product.

On the subgoal `forall x y z : nat, x = y -> y = x` the tactic `intros until 1` is equivalent to `intros x y z` as the product on `z` can be rewritten as a non-dependent product: `forall x y : nat, nat -> x = y -> y = x`.

Error: `No such hypothesis in current goal.`

This happens when *num* is 0 or is greater than the number of non-dependent products of the goal.

Variant: `intro` *ident*$_1$$^?$ `after` *ident*$_2$

Variant: `intro` *ident*$_1$$^?$ `before` *ident*$_2$

Variant: `intro` *ident*$_1$$^?$ `at top`

Variant: `intro` *ident*$_1$$^?$ `at bottom`

These tactics apply `intro` *ident*$_1$$^?$ and move the freshly introduced hypothesis respectively after the hypothesis *ident*$_2$, before the hypothesis *ident*$_2$, at the top of the local context, or at the bottom of the local context. All hypotheses on which the new hypothesis depends are moved too so as to respect the order of dependencies between hypotheses. It is equivalent to `intro` *ident*$_1$$^?$ followed by the appropriate call to *move ... after ...*, *move ... before ...*, *move ... at top*, or *move ... at bottom*.

Note: `intro at bottom` is a synonym for `intro` with no argument.

Error: `No such hypothesis:` *ident*`.`

`intros` *intropattern_list*

Introduces one or more variables or hypotheses from the goal by matching the intro patterns. See the description in *Intro patterns*.

eintros *intropattern_list*

Works just like *intros ...* except that it creates existential variables for any unresolved variables rather than failing.

clear *ident*

This tactic erases the hypothesis named *ident* in the local context of the current goal. As a consequence, *ident* is no more displayed and no more usable in the proof development.

Error: **No such hypothesis.**

Error: *ident* **is used in the conclusion.**

Error: *ident* **is used in the hypothesis** *ident*.

Variant: **clear** *ident*+

This is equivalent to `clear` *ident*. `...` `clear` *ident*.

Variant: **clear -** *ident*+

This variant clears all the hypotheses except the ones depending in the hypotheses named *ident*+ and in the goal.

Variant: **clear**

This variants clears all the hypotheses except the ones the goal depends on.

Variant: **clear dependent** *ident*

This clears the hypothesis *ident* and all the hypotheses that depend on it.

Variant: **clearbody** *ident*+

This tactic expects *ident*+ to be local definitions and clears their respective bodies. In other words, it turns the given definitions into assumptions.

Error: *ident* **is not a local definition.**

revert *ident*+

This applies to any goal with variables *ident*+. It moves the hypotheses (possibly defined) to the goal, if this respects dependencies. This tactic is the inverse of *intro*.

Error: **No such hypothesis.**

Error: *ident*$_1$ **is used in the hypothesis** *ident*$_2$.

Variant: **revert dependent** *ident*

This moves to the goal the hypothesis *ident* and all the hypotheses that depend on it.

move *ident*$_1$ **after** *ident*$_2$

This moves the hypothesis named *ident*$_1$ in the local context after the hypothesis named *ident*$_2$, where "after" is in reference to the direction of the move. The proof term is not changed.

If *ident*$_1$ comes before *ident*$_2$ in the order of dependencies, then all the hypotheses between *ident*$_1$ and *ident*$_2$ that (possibly indirectly) depend on *ident*$_1$ are moved too, and all of them are thus moved after *ident*$_2$ in the order of dependencies.

If *ident*$_1$ comes after *ident*$_2$ in the order of dependencies, then all the hypotheses between *ident*$_1$ and *ident*$_2$ that (possibly indirectly) occur in the type of *ident*$_1$ are moved too, and all of them are thus moved before *ident*$_2$ in the order of dependencies.

Variant: **move** *ident*$_1$ **before** *ident*$_2$

This moves *ident*$_1$ towards and just before the hypothesis named *ident*$_2$. As for *move ... after ...*, dependencies over *ident*$_1$ (when *ident*$_1$ comes before *ident*$_2$ in the order of dependencies) or in the type of *ident*$_1$ (when *ident*$_1$ comes after *ident*$_2$ in the order of dependencies) are moved too.

Variant: `move` *ident* `at top`

This moves *ident* at the top of the local context (at the beginning of the context).

Variant: `move` *ident* `at bottom`

This moves *ident* at the bottom of the local context (at the end of the context).

Error: `No such hypothesis.`

Error: `Cannot move` *ident*$_1$ `after` *ident*$_2$`: it occurs in the type of` *ident*$_2$`.`

Error: `Cannot move` *ident*$_1$ `after` *ident*$_2$`: it depends on` *ident*$_2$`.`

Example

```
Goal forall x :nat, x = 0 -> forall z y:nat, y=y-> 0=x.
    1 subgoal

      ============================
      forall x : nat, x = 0 -> nat -> forall y : nat, y = y -> 0 = x

intros x H z y H0.
    1 subgoal

      x : nat
      H : x = 0
      z, y : nat
      H0 : y = y
      ============================
      0 = x

move x after H0.
    1 subgoal

      z, y : nat
      H0 : y = y
      x : nat
      H : x = 0
      ============================
      0 = x

Undo.
    1 subgoal

      x : nat
      H : x = 0
      z, y : nat
      H0 : y = y
      ============================
      0 = x

move x before H0.
    1 subgoal

      z, y, x : nat
      H : x = 0
      H0 : y = y
```

(continues on next page)

(continued from previous page)

```
      ============================
      0 = x

Undo.
    1 subgoal

      x : nat
      H : x = 0
      z, y : nat
      H0 : y = y
      ============================
      0 = x

move H0 after H.
    1 subgoal

      x, y : nat
      H0 : y = y
      H : x = 0
      z : nat
      ============================
      0 = x

Undo.
    1 subgoal

      x : nat
      H : x = 0
      z, y : nat
      H0 : y = y
      ============================
      0 = x

move H0 before H.
    1 subgoal

      x : nat
      H : x = 0
      y : nat
      H0 : y = y
      z : nat
      ============================
      0 = x
```

rename *ident*$_1$ **into** *ident*$_2$

This renames hypothesis *ident*$_1$ into *ident*$_2$ in the current context. The name of the hypothesis in the proof-term, however, is left unchanged.

Variant: rename *ident*$_i$ **into** *ident*$_j$ $^{+}_{,}$

This renames the variables *ident*$_i$ into *ident*$_j$ in parallel. In particular, the target identifiers may contain identifiers that exist in the source context, as long as the latter are also renamed by the same tactic.

Error: No such hypothesis.

Error: *ident* **is already used.**

`set (`*`ident`*` := `*`term`*`)`

This replaces *term* by *ident* in the conclusion of the current goal and adds the new definition *ident* `:=` *term* to the local context.

If *term* has holes (i.e. subexpressions of the form "_"), the tactic first checks that all subterms matching the pattern are compatible before doing the replacement using the leftmost subterm matching the pattern.

Error: `The variable` *`ident`* `is already defined.`

Variant: `set (`*`ident`*` := `*`term`*`) in `*`goal_occurrences`*

This notation allows specifying which occurrences of *term* have to be substituted in the context. The `in` *goal_occurrences* clause is an occurrence clause whose syntax and behavior are described in *goal occurrences*.

Variant: `set (`*`ident binders`*` := `*`term`*`)` `in `*`goal_occurrences`*$^?$

This is equivalent to `set (`*`ident`*` := fun `*`binders`*` => `*`term`*`)` `in `*`goal_occurrences`*$^?$.

Variant: `set `*`term`* `in `*`goal_occurrences`*$^?$

This behaves as `set (`*`ident`*` := `*`term`*`)` `in `*`goal_occurrences`*$^?$ but *ident* is generated by Coq.

Variant: `eset (`*`ident`* *`binders`*$^?$ `:= `*`term`*`)` `in `*`goal_occurrences`*$^?$

Variant: `eset `*`term`* `in `*`goal_occurrences`*$^?$

While the different variants of *set* expect that no existential variables are generated by the tactic, *eset* removes this constraint. In practice, this is relevant only when *eset* is used as a synonym of *epose*, i.e. when the *term* does not occur in the goal.

`remember `*`term`*` as `*`ident`*$_1$ `eqn:`*`naming_intropattern`*$^?$

This behaves as `set (`*`ident`*` := `*`term`*`) in *`, using a logical (Leibniz's) equality instead of a local definition. Use *naming_intropattern* to name or split up the new equation.

Variant: `remember `*`term`*` as `*`ident`*$_1$ `eqn:`*`naming_intropattern`*$^?$ `in `*`goal_occurrences`*

This is a more general form of *remember* that remembers the occurrences of *term* specified by an occurrence set.

Variant: `eremember `*`term`*` as `*`ident`*$_1$ `eqn:`*`naming_intropattern`*$^?$ `in `*`goal_occurrences`*$^?$

While the different variants of *remember* expect that no existential variables are generated by the tactic, *eremember* removes this constraint.

`pose (`*`ident`*` := `*`term`*`)`

This adds the local definition *ident* `:=` *term* to the current context without performing any replacement in the goal or in the hypotheses. It is equivalent to `set (`*`ident`*` := `*`term`*`) in |-`.

Variant: `pose (`*`ident binders`*` := `*`term`*`)`

This is equivalent to `pose (`*`ident`*` := fun `*`binders`*` => `*`term`*`)`.

Variant: `pose `*`term`*

This behaves as `pose (`*`ident`*` := `*`term`*`)` but *ident* is generated by Coq.

Variant: `epose (`*`ident`* *`binders`*$^?$ `:= `*`term`*`)`

Variant: `epose `*`term`*

While the different variants of *pose* expect that no existential variables are generated by the tactic, *epose* removes this constraint.

`decompose [`*`qualid`*$^{+}$`]` *`term`*

This tactic recursively decomposes a complex proposition in order to obtain atomic ones.

Example

```
Goal 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

intros A B C H; decompose [and or] H.
    3 subgoals

      A, B, C : Prop
      H : A /\ B /\ C \/ B /\ C \/ C /\ A
      H1 : A
      H0 : B
      H3 : C
      ============================
      C

    subgoal 2 is:
     C
    subgoal 3 is:
     C

all: assumption.
    No more subgoals.

Qed.
```

Note: *`decompose`* does not work on right-hand sides of implications or products.

Variant: `decompose sum` *`term`*

This decomposes sum types (like `or`).

Variant: `decompose record` *`term`*

This decomposes record types (inductive types with one constructor, like `and` and `exists` and those defined with the *`Record`* command.

5.3.4 Controlling the proof flow

`assert (`*`ident`*` : `*`type`*`)`

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`[2]. The subgoal `U` comes first in the list of subgoals remaining to prove.

Error: `Not a proposition or a type.`

Arises when the argument *`type`* is neither of type `Prop`, `Set` nor `Type`.

[2] This corresponds to the cut rule of sequent calculus.

Variant: `assert` *type*
This behaves as `assert (`*ident* `:` *type*`)` but *ident* is generated by Coq.

Variant: `assert` *type* `by` *tactic*
This tactic behaves like *assert* but applies tactic to solve the subgoals generated by assert.

Error: `Proof is not complete.`

Variant: `assert` *type* `as` *simple_intropattern*
If `simple_intropattern` is an intro pattern (see *Intro patterns*), the hypothesis is named after this introduction pattern (in particular, if `simple_intropattern` is *ident*, the tactic behaves like `assert (`*ident* `:` *type*`)`). If `simple_intropattern` is an action introduction pattern, the tactic behaves like `assert` *type* followed by the action done by this introduction pattern.

Variant: `assert` *type* `as` *simple_intropattern* `by` *tactic*
This combines the two previous variants of *assert*.

Variant: `assert (`*ident* `:=` *term*`)`
This behaves as `assert (`*ident* `:` *type*`) by exact` *term* where *type* is the type of *term*. This is equivalent to using *pose proof*. If the head of term is *ident*, the tactic behaves as *specialize*.

Error: `Variable` *ident* `is already declared.`

Variant: `eassert` *type* `as` *simple_intropattern* `by` *tactic*
While the different variants of *assert* expect that no existential variables are generated by the tactic, *eassert* removes this constraint. This lets you avoid specifying the asserted statement completely before starting to prove it.

Variant: `pose proof` *term* `as` *simple_intropattern*?

This tactic behaves like `assert` *type* `as` *simple_intropattern*? `by exact` *term* where *type* is the type of *term*. In particular, `pose proof` *term* `as` *ident* behaves as `assert (`*ident* `:=` *term*`)` and `pose proof` *term* `as` *simple_intropattern* is the same as applying the *simple_intropattern* to *term*.

Variant: `epose proof` *term* `as` *simple_intropattern*?
While *pose proof* expects that no existential variables are generated by the tactic, *epose proof* removes this constraint.

Variant: `enough (`*ident* `:` *type*`)`
This adds a new hypothesis of name *ident* asserting *type* to the goal the tactic *enough* is applied to. A new subgoal stating *type* is inserted after the initial goal rather than before it as *assert* would do.

Variant: `enough` *type*
This behaves like `enough (`*ident* `:` *type*`)` with the name *ident* of the hypothesis generated by Coq.

Variant: `enough` *type* `as` *simple_intropattern*
This behaves like `enough` *type* using *simple_intropattern* to name or destruct the new hypothesis.

Variant: `enough (`*ident* `:` *type*`) by` *tactic*

Variant: `enough` *type* `as` *simple_intropattern*? `by` *tactic*
This behaves as above but with `tactic` expected to solve the initial goal after the extra assumption *type* is added and possibly destructed. If the `as` *simple_intropattern* clause generates more than one subgoal, `tactic` is applied to all of them.

Variant: `eenough` *type* `as` *simple_intropattern*? `by` *tactic*?

Variant: `eenough (`*ident* `:` *type*`)` `by` *tactic*?
While the different variants of *enough* expect that no existential variables are generated by the tactic,

eenough removes this constraint.

Variant: `cut` *type*

This tactic applies to any goal. It implements the non-dependent case of the "App" rule given in *Typing rules*. (This is Modus Ponens inference rule.) `cut U` transforms the current goal `T` into the two following subgoals: `U -> T` and `U`. The subgoal `U -> T` comes first in the list of remaining subgoal to prove.

Variant: `specialize (`*ident* *term**`)` `as` *simple_intropattern*$^?$

Variant: `specialize` *ident* `with` *bindings_list* `as` *simple_intropattern*$^?$

This tactic works on local hypothesis *ident*. The premises of this hypothesis (either universal quantifications or non-dependent implications) are instantiated by concrete terms coming either from arguments *term** or from a *bindings list*. In the first form the application to *term** can be partial. The first form is equivalent to `assert (`*ident* `:=` *ident* *term**`)`. In the second form, instantiation elements can also be partial. In this case the uninstantiated arguments are inferred by unification if possible or left quantified in the hypothesis otherwise. With the `as` clause, the local hypothesis *ident* is left unchanged and instead, the modified hypothesis is introduced as specified by the *simple_intropattern*. The name *ident* can also refer to a global lemma or hypothesis. In this case, for compatibility reasons, the behavior of *specialize* is close to that of *generalize*: the instantiated statement becomes an additional premise of the goal. The `as` clause is especially useful in this case to immediately introduce the instantiated statement as a local hypothesis.

Error: *ident* **`is used in hypothesis`** *ident*.

Error: *ident* **`is used in conclusion.`**

`generalize` *term*

This tactic applies to any goal. It generalizes the conclusion with respect to some term.

Example

```
Show.
    1 subgoal

      x, y : nat
      ============================
      0 <= x + y + y

generalize (x + y + y).
    1 subgoal

      x, y : nat
      ============================
      forall n : nat, 0 <= n
```

If the goal is `G` and `t` is a subterm of type `T` in the goal, then `generalize t` replaces the goal by `forall (x:T), G` where `G` is obtained from `G` by replacing all occurrences of `t` by `x`. The name of the variable (here `n`) is chosen based on `T`.

Variant: `generalize` *term*$^+$

This is equivalent to `generalize` *term*; `...` ; `generalize` *term*. Note that the sequence of term $_i$'s are processed from n to 1.

Variant: `generalize` *term* `at` *num*$^+$

This is equivalent to `generalize` *term* but it generalizes only over the specified occurrences of *term* (counting from left to right on the expression printed using the *Printing All* flag).

Variant: `generalize` *term* `as` *ident*

This is equivalent to `generalize` *term* but it uses *ident* to name the generalized hypothesis.

Variant: `generalize` *term* `at` *num* + `as` *ident* + ,

This is the most general form of `generalize` that combines the previous behaviors.

Variant: `generalize dependent` *term*

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

`evar (`*ident* `:` *term*`)`

The `evar` tactic creates a new local definition named *ident* with type *term* in the context. The body of this binding is a fresh existential variable.

`instantiate (`*ident* `:=` *term* `)`

The instantiate tactic refines (see *refine*) an existential variable *ident* with the term *term*. It is equivalent to `only [ident]: refine` *term* (preferred alternative).

Note: To be able to refer to an existential variable by name, the user must have given the name explicitly (see *Existential variables*).

Note: When you are referring to hypotheses which you did not name explicitly, be aware that Coq may make a different decision on how to name the variable in the current goal and in the context of the existential variable. This can lead to surprising behaviors.

Variant: `instantiate (`*num* `:=` *term*`)`

This variant allows to refer to an existential variable which was not named by the user. The *num* argument is the position of the existential variable from right to left in the goal. Because this variant is not robust to slight changes in the goal, its use is strongly discouraged.

Variant: `instantiate (` *num* `:=` *term* `) in` *ident*

Variant: `instantiate (` *num* `:=` *term* `) in ( value of` *ident* `)`

Variant: `instantiate (` *num* `:=` *term* `) in ( type of` *ident* `)`

These allow to refer respectively to existential variables occurring in a hypothesis or in the body or the type of a local definition.

Variant: `instantiate`

Without argument, the instantiate tactic tries to solve as many existential variables as possible, using information gathered from other tactics in the same tactical. This is automatically done after each complete tactic (i.e. after a dot in proof mode), but not, for example, between each tactic when they are sequenced by semicolons.

`admit`

This tactic allows temporarily skipping a subgoal so as to progress further in the rest of the proof. A proof containing admitted goals cannot be closed with *Qed* but only with *Admitted*.

Variant: `give_up`

Synonym of *admit*.

`absurd` *term*

This tactic applies to any goal. The argument term is any proposition `P` of type `Prop`. This tactic applies False elimination, that is it deduces the current goal from False, and generates as subgoals `P` and P. It is very useful in proofs by cases, where some cases are impossible. In most cases, P or P is one of the hypotheses of the local context.

contradiction

This tactic applies to any goal. The contradiction tactic attempts to find in the current context (after all intros) a hypothesis that is equivalent to an empty inductive type (e.g. `False`), to the negation of a singleton inductive type (e.g. `True` or `x=x`), or two contradictory hypotheses.

Error: `No such assumption.`

Variant: `contradiction` *ident*

The proof of False is searched in the hypothesis named *ident*.

`contradict` *ident*

This tactic allows manipulating negated hypothesis and goals. The name *ident* should correspond to a hypothesis. With `contradict H`, the current goal and context is transformed in the following way:

- H:¬A B becomes A
- H:¬A ¬B becomes H: B A
- H: A B becomes ¬A
- H: A ¬B becomes H: B ¬A

`exfalso`

This tactic implements the "ex falso quodlibet" logical principle: an elimination of False is performed on the current goal, and the user is then required to prove that False is indeed provable in the current context. This tactic is a macro for `elimtype False`.

5.3.5 Case analysis and induction

The tactics presented in this section implement induction or case analysis on inductive or co-inductive objects (see *Inductive Definitions*).

`destruct` *term*

This tactic applies to any goal. The argument *term* must be of inductive or co-inductive type and the tactic generates subgoals, one for each possible form of *term*, i.e. one for each constructor of the inductive or co-inductive type. Unlike *induction*, no induction hypothesis is generated by *destruct*.

Variant: `destruct` *ident*

If *ident* denotes a quantified variable of the conclusion of the goal, then `destruct` *ident* behaves as `intros until` *ident*; `destruct` *ident*. If *ident* is not anymore dependent in the goal after application of *destruct*, it is erased (to avoid erasure, use parentheses, as in `destruct (`*ident*`)`).

If *ident* is a hypothesis of the context, and *ident* is not anymore dependent in the goal after application of *destruct*, it is erased (to avoid erasure, use parentheses, as in `destruct (`*ident*`)`).

Variant: `destruct` *num*

`destruct` *num* behaves as `intros until` *num* followed by destruct applied to the last introduced hypothesis.

Note: For destruction of a numeral, use syntax `destruct (`*num*`)` (not very interesting anyway).

Variant: `destruct` *pattern*

The argument of *destruct* can also be a pattern of which holes are denoted by "_". In this case, the tactic checks that all subterms matching the pattern in the conclusion and the hypotheses are compatible and performs case analysis using this subterm.

Variant: `destruct` *term*$^{+}_{,}$

This is a shortcut for `destruct` *term*`; ...; destruct` *term*.

Variant: `destruct` *term* `as` *or_and_intropattern_loc*

This behaves as `destruct` *term* but uses the names in *or_and_intropattern_loc* to name the variables introduced in the context. The *or_and_intropattern_loc* must have the form `[p11 ... p1n | ... | pm1 ... pmn ]` 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 `pi1 ... pin` in order. If there are not enough names, *destruct* invents names for the remaining variables to introduce. More generally, the `pij` can be any introduction pattern (see *intros*). This provides a concise notation for chaining destruction of a hypothesis.

Variant: `destruct` *term* `eqn:`*naming_intropattern*

This behaves as `destruct` *term* but adds an equation between *term* and the value that it takes in each of the possible cases. The name of the equation is specified by *naming_intropattern* (see *intros*), in particular `?` can be used to let Coq generate a fresh name.

Variant: `destruct` *term* `with` *bindings_list*

This behaves like `destruct` *term* providing explicit instances for the dependent premises of the type of *term*.

Variant: `edestruct` *term*

This tactic behaves like `destruct` *term* except that it does not fail if the instance of a dependent premises of the type of *term* is not inferable. Instead, the unresolved instances are left as existential variables to be inferred later, in the same way as *eapply* does.

Variant: `destruct` *term* `using` *term* [`with` *bindings_list*]$^{?}$

This is synonym of `induction` *term* `using` *term* [`with` *bindings_list*]$^{?}$.

Variant: `destruct` *term* `in` *goal_occurrences*

This syntax is used for selecting which occurrences of *term* the case analysis has to be done on. The `in` *goal_occurrences* clause is an occurrence clause whose syntax and behavior is described in *occurrences sets*.

Variant: `destruct` *term* [`with` *bindings_list*]$^{?}$ [`as` *or_and_intropattern_loc*]$^{?}$ [`eqn:`*naming_intropattern*

Variant: `edestruct` *term* [`with` *bindings_list*]$^{?}$ [`as` *or_and_intropattern_loc*]$^{?}$ [`eqn:`*naming_intropattern*

These are the general forms of *destruct* and *edestruct*. They combine the effects of the `with`, `as`, `eqn:`, `using`, and `in` clauses.

`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.

Variant: `case` *term* `with` *bindings_list*

Analogous to `elim` *term* `with` *bindings_list* above.

Variant: `ecase` *term* [`with` *bindings_list*]$^{?}$

In case the type of *term* has dependent premises, or dependent premises whose values are not inferable from the `with` *bindings_list* clause, `ecase` turns them into existential variables to be resolved later on.

Variant: `simple destruct` *ident*

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

Variant: 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.

Variant: case_eq *term*

The tactic `case_eq` is a variant of the `case` tactic that allows to perform case analysis on a term without completely forgetting its original form. This is done by generating equalities between the original form of the term and the outcomes of the case analysis.

induction *term*

This tactic applies to any goal. The argument *term* must be of inductive type and the tactic `induction` generates subgoals, one for each possible form of *term*, i.e. one for each constructor of the inductive type.

If the argument is dependent in either the conclusion or some hypotheses of the goal, the argument is replaced by the appropriate constructor form in each of the resulting subgoals and induction hypotheses are added to the local context using names whose prefix is **IH**.

There are particular cases:

- If term is an identifier *ident* denoting a quantified variable of the conclusion of the goal, then inductionident behaves as `intros until` *ident*; `induction` *ident*. If *ident* is not anymore dependent in the goal after application of `induction`, it is erased (to avoid erasure, use parentheses, as in `induction (`*ident*`)`).
- If *term* is a *num*, then `induction` *num* behaves as `intros until` *num* followed by `induction` applied to the last introduced hypothesis.

  Note: For simple induction on a numeral, use syntax induction (num) (not very interesting anyway).

- In case term is a hypothesis *ident* of the context, and *ident* is not anymore dependent in the goal after application of `induction`, it is erased (to avoid erasure, use parentheses, as in `induction (`*ident*`)`).
- The argument *term* can also be a pattern of which holes are denoted by "_". In this case, the tactic checks that all subterms matching the pattern in the conclusion and the hypotheses are compatible and performs induction using this subterm.

Example

```
Lemma induction_test : forall n:nat, n = n -> n <= n.
    1 subgoal

      ============================
      forall n : nat, n = n -> n <= n

intros n H.
    1 subgoal

      n : nat
      H : n = n
      ============================
      n <= n

induction n.
    2 subgoals
```

(continues on next page)

(continued from previous page)

```
      H : 0 = 0
      ============================
      0 <= 0

    subgoal 2 is:
     S n <= S n

exact (le_n 0).
    1 subgoal

      n : nat
      H : S n = S n
      IHn : n = n -> n <= n
      ============================
      S n <= S n
```

Error: `Not an inductive product.`

Error: `Unable to find an instance for the variables` *ident* `...` *ident*`.`

Use in this case the variant *elim ... with* below.

Variant: `induction` *term* `as` *or_and_intropattern_loc*

This behaves as *induction* but uses the names in *or_and_intropattern_loc* to name the variables introduced in the context. The *or_and_intropattern_loc* must typically be of the form `[` p_{11} `...` p_{1n} `|` `...` `|` p_{m1} `...` p_{mn} `]` 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} `...` p_{in} in order. If there are not enough names, induction invents names for the remaining variables to introduce. More generally, the p_{ij} can be any disjunctive/conjunctive introduction pattern (see *intros ...*). For instance, for an inductive type with one constructor, the pattern notation `(`p_1 `,` `...` `,` p_n `)` can be used instead of `[` p_1 `...` p_n `]`.

Variant: `induction` *term* `with` *bindings_list*

This behaves like *induction* providing explicit instances for the premises of the type of `term` (see *bindings list*).

Variant: `einduction` *term*

This tactic behaves like *induction* except that it does not fail if some dependent premise of the type of *term* is not inferable. Instead, the unresolved premises are posed as existential variables to be inferred later, in the same way as *eapply* does.

Variant: `induction` *term* `using` *term*

This behaves as *induction* but using *term* as induction scheme. It does not expect the conclusion of the type of the first *term* to be inductive.

Variant: `induction` *term* `using` *term* `with` *bindings_list*

This behaves as *induction ... using ...* but also providing instances for the premises of the type of the second *term*.

Variant: `induction` *term*+, `using` *qualid*

This syntax is used for the case *qualid* denotes an induction principle with complex predicates as the induction principles generated by `Function` or `Functional Scheme` may be.

Variant: `induction` *term* `in` *goal_occurrences*

This syntax is used for selecting which occurrences of *term* the induction has to be carried on. The `in` *goal_occurrences* clause is an occurrence clause whose syntax and behavior is described in *occurrences* *sets*. If variables or hypotheses not mentioning `term` in their type are listed in `goal_occurrences`, those are generalized as well in the statement to prove.

Example

```
Lemma comm x y : x + y = y + x.
    1 subgoal

      x, y : nat
      ============================
      x + y = y + x

induction y in x |-   *.
    2 subgoals

      x : nat
      ============================
      x + 0 = 0 + x

    subgoal 2 is:
     x + S y = S y + x

Show 2.
    subgoal 2 is:

      x, y : nat
      IHy : forall x : nat, x + y = y + x
      ============================
      x + S y = S y + x
```

Variant: `induction term with bindings_list as or_and_intropattern_loc using term with bindings_list in g`
Variant: `einduction term with bindings_list as or_and_intropattern_loc using term with bindings_list in`

These are the most general forms of *induction* and *einduction*. It combines the effects of the with, as, using, and in clauses.

Variant: `elim term`

This is a more basic induction tactic. Again, the type of the argument `term` must be an inductive type. Then, according to the type of the goal, the tactic `elim` chooses the appropriate destructor and applies it as the tactic *apply* would do. For instance, if the proof context contains `n:nat` and the current goal is `T` of type `Prop`, then `elim n` is equivalent to `apply nat_ind with (n:=n)`. The tactic `elim` does not modify the context of the goal, neither introduces the induction loading into the context of hypotheses. More generally, `elim term` also works when the type of `term` is a statement with premises and whose conclusion is inductive. In that case the tactic performs induction on the conclusion of the type of `term` and leaves the non-dependent premises of the type as subgoals. In the case of dependent products, the tactic tries to find an instance for which the elimination lemma applies and fails otherwise.

Variant: `elim term with bindings_list`

Allows to give explicit instances to the premises of the type of `term` (see *bindings list*).

Variant: `eelim term`

In case the type of `term` has dependent premises, this turns them into existential variables to be resolved later on.

Variant: `elim term using term`
Variant: `elim term using term with bindings_list`

Allows the user to give explicitly an induction principle `term` that is not the standard one for the underlying inductive type of *term*. The *bindings_list* clause allows instantiating premises of the type of *term*.

Variant: `elim` *term* `with` *bindings_list* `using` *term* `with` *bindings_list*

Variant: `eelim` *term* `with` *bindings_list* `using` *term* `with` *bindings_list*

These are the most general forms of *elim* and *eelim*. It combines the effects of the `using` clause and of the two uses of the `with` clause.

Variant: `elimtype` *type*

The argument *type* 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` that does not occur in the goal, then `elim t` is equivalent to `elimtype I; 2:exact t.`

Variant: `simple induction` *ident*

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

Variant: `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.

`double induction` *ident* *ident*

This tactic is deprecated and should be replaced by `induction` *ident*`; induction` *ident* (or `induction` *ident* `; destruct` *ident* depending on the exact needs).

Variant: `double induction` num_1 num_2

This tactic is deprecated and should be replaced by `induction num1; induction num3` where `num3` is the result of `num2 - num1`

`dependent induction` *ident*

The *experimental* tactic dependent induction performs induction- inversion on an instantiated inductive predicate. One needs to first require the Coq.Program.Equality module to use this tactic. The tactic is based on the BasicElim tactic by Conor McBride [McB00] and the work of Cristina Cornes around inversion [CT95]. From an instantiated inductive predicate and a goal, it generates an equivalent goal where the hypothesis has been generalized over its indexes which are then constrained by equalities to be the right instances. This permits to state lemmas without resorting to manually adding these equalities and still get enough information in the proofs.

Example

```
Lemma lt_1_r : forall n:nat, n < 1 -> n = 0.
    1 subgoal

      ============================
      forall n : nat, n < 1 -> n = 0

intros n H ; induction H.
    2 subgoals

      n : nat
      ============================
      n = 0

    subgoal 2 is:
     n = 0
```

Here we did not get any information on the indexes to help fulfill this proof. The problem is that, when we use the `induction` tactic, we lose information on the hypothesis instance, notably that the second argument is 1 here. Dependent induction solves this problem by adding the corresponding equality to the context.

```
Require Import Coq.Program.Equality.
Lemma lt_1_r : forall n:nat, n < 1 -> n = 0.
    1 subgoal

      ============================
      forall n : nat, n < 1 -> n = 0

intros n H ; dependent induction H.
    2 subgoals

      ============================
      0 = 0

    subgoal 2 is:
     n = 0
```

The subgoal is cleaned up as the tactic tries to automatically simplify the subgoals with respect to the generated equalities. In this enriched context, it becomes possible to solve this subgoal.

```
reflexivity.
    1 subgoal

      n : nat
      H : S n <= 0
      IHle : 0 = 1 -> n = 0
      ============================
      n = 0
```

Now we are in a contradictory context and the proof can be solved.

```
inversion H.
    No more subgoals.
```

This technique works with any inductive predicate. In fact, the `dependent induction` tactic is just a wrapper around the `induction` tactic. One can make its own variant by just writing a new tactic based on the definition found in `Coq.Program.Equality`.

Variant: `dependent induction` *ident* `generalizing` *ident*+

This performs dependent induction on the hypothesis *ident* but first generalizes the goal by the given variables so that they are universally quantified in the goal. This is generally what one wants to do with the variables that are inside some constructors in the induction hypothesis. The other ones need not be further generalized.

Variant: `dependent destruction` *ident*

This performs the generalization of the instance *ident* but uses `destruct` instead of induction on the generalized hypothesis. This gives results equivalent to `inversion` or `dependent inversion` if the hypothesis is dependent.

See also the larger example of *dependent induction* and an explanation of the underlying technique.

`function induction (`*qualid* *term*+`)`

The tactic functional induction performs case analysis and induction following the definition of a function. It makes use of a principle generated by `Function` (see *Advanced recursive functions*) or

Functional Scheme (see *Generation of induction principles with Functional Scheme*). Note that this tactic is only available after a Require Import FunInd.

Example

```
Require Import FunInd.
    [Loading ML file extraction_plugin.cmxs ... done]
    [Loading ML file recdef_plugin.cmxs ... done]

Functional Scheme minus_ind := Induction for minus Sort Prop.
    sub_equation is defined
    minus_ind is defined

Check minus_ind.
    minus_ind
         : forall P : nat -> nat -> nat -> Prop,
           (forall n m : nat, n = 0 -> P 0 m n) ->
           (forall n m k : nat, n = S k -> m = 0 -> P (S k) 0 n) ->
           (forall n m k : nat,
            n = S k ->
            forall l : nat, m = S l -> P k l (k - l) -> P (S k) (S l) (k - l)) ->
           forall n m : nat, P n m (n - m)

Lemma le_minus (n m:nat) : n - m <= n.
    1 subgoal

      n, m : nat
      ============================
      n - m <= n

functional induction (minus n m) using minus_ind; simpl; auto.
    No more subgoals.

Qed.
```

Note: (*qualid* *term*+) must be a correct full application of *qualid*. In particular, the rules for implicit arguments are the same as usual. For example use *qualid* if you want to write implicit arguments explicitly.

Note: Parentheses around *qualid* *term*+ are not mandatory and can be skipped.

Note: functional induction (f x1 x2 x3) is actually a wrapper for induction x1, x2, x3, (f x1 x2 x3) using *qualid* followed by a cleaning phase, where *qualid* is the induction principle registered for f (by the Function (see *Advanced recursive functions*) or Functional Scheme (see *Generation of induction principles with Functional Scheme*) command) corresponding to the sort of the goal. Therefore functional induction may fail if the induction scheme *qualid* is not defined. See also *Advanced recursive functions* for the function terms accepted by Function.

Note: There is a difference between obtaining an induction scheme for a function by using Function (see *Advanced recursive functions*) and by using Functional Scheme after a normal definition using Fixpoint or Definition. See *Advanced recursive functions* for details.

See also:

Advanced recursive functions, *Generation of induction principles with Functional Scheme* and *inversion*

Error: `Cannot find induction information on qualid.`

Error: `Not the right number of induction arguments.`

Variant: `functional induction (qualid term+) as simple_intropattern using term with bindings_list`

Similarly to *induction* and *elim*, this allows giving explicitly the name of the introduced variables, the induction principle, and the values of dependent premises of the elimination scheme, including *predicates* for mutual induction when *qualid* is part of a mutually recursive definition.

`discriminate term`

This tactic proves any goal from an assumption stating that two structurally different *term*s of an inductive set are equal. For example, from `(S (S O))=(S O)` we can derive by absurdity any proposition.

The argument *term* is assumed to be a proof of a statement of conclusion `term = term` with the two terms being elements of an inductive set. To build the proof, the tactic traverses the normal forms[3] of the terms looking for a couple of subterms `u` and `w` (`u` subterm of the normal form of *term* and `w` subterm of the normal form of *term*), 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.

Note: The syntax `discriminate ident` can be used to refer to a hypothesis quantified in the goal. In this case, the quantified hypothesis whose name is *ident* is first introduced in the local context using `intros until ident`.

Error: `No primitive equality found.`

Error: `Not a discriminable equality.`

Variant: `discriminate num`

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

Variant: `discriminate term with bindings_list`

This does the same thing as `discriminate term` but using the given bindings to instantiate parameters or hypotheses of *term*.

Variant: `ediscriminate num`

Variant: `ediscriminate term [with bindings_list]?`

This works the same as *discriminate* but if the type of *term*, or the type of the hypothesis referred to by *num*, has uninstantiated parameters, these parameters are left as existential variables.

Variant: `discriminate`

This behaves like `discriminate ident` if ident is the name of an hypothesis to which `discriminate` is applicable; if the current goal is of the form `term <> term`, this behaves as `intro ident; discriminate ident`.

Error: `No discriminable equalities.`

[3] Reminder: opaque constants will not be expanded by δ reductions.

injection *term*

The injection tactic exploits the property that constructors of inductive types are injective, i.e. that if c is a constructor of an inductive type and c t_1 and c t_2 are equal then t_1 and t_2 are equal too.

If *term* is a proof of a statement of conclusion *term* = *term*, then *injection* applies the injectivity of constructors as deep as possible to derive the equality of all the subterms of *term* and *term* at positions where the terms start to differ. For example, from (S p, S n) = (q, S (S m)) we may derive S p = q and n = S m. For this tactic to work, the terms should be typed with an inductive type and they should be neither convertible, nor having a different head constructor. If these conditions are satisfied, the tactic derives the equality of all the subterms at positions where they differ and adds them as antecedents to the conclusion of the current goal.

Example

Consider the following goal:

```
Inductive list : Set :=
| nil : list
| cons : nat -> list -> list.
Parameter P : list -> Prop.
Goal forall l n, P nil -> cons n l = cons 0 nil -> P l.

intros.
    1 subgoal

      l : list
      n : nat
      H : P nil
      H0 : cons n l = cons 0 nil
      ============================
      P l

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 an equality in a sigma type whenever the injected object has a dependent type P with its two instances in different types (P t_1 ... t_n) and (P u_1 ... u_n). If t_1 and u_1 are the same and have for type an inductive type for which a decidable equality has been declared using the command *Scheme Equality* (see *Generation of induction principles with Scheme*), the use of a sigma type is avoided.

Note: If some quantified hypothesis of the goal is named *ident*, then injection *ident* first introduces the hypothesis in the local context using intros until *ident*.

Error: Not a projectable equality but a discriminable one.

Error: Nothing to do, it is an equality between convertible terms.

Error: Not a primitive equality.

Error: `Nothing to inject.`

Variant: `injection` *num*

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

Variant: `injection` *term* `with` *bindings_list*

This does the same as `injection` *term* but using the given bindings to instantiate parameters or hypotheses of *term*.

Variant: `einjection` *num*

Variant: `einjection` *term* `with` *bindings_list* ?

This works the same as `injection` but if the type of *term*, or the type of the hypothesis referred to by *num*, has uninstantiated parameters, these parameters are left as existential variables.

Variant: `injection`

If the current goal is of the form *term* `<>` *term* , this behaves as `intro` *ident*`;` `injection` *ident*.

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

Variant: `injection` *term* `with` *bindings_list* ? `as` *simple_intropattern* +

Variant: `injection` *num* `as` *simple_intropattern* +

Variant: `injection as` *simple_intropattern* +

Variant: `einjection` *term* `with` *bindings_list* ? `as` *simple_intropattern* +

Variant: `einjection` *num* `as` *simple_intropattern* +

Variant: `einjection as` *simple_intropattern* +

These variants apply `intros` *simple_intropattern* + after the call to *injection* or *einjection* so that all equalities generated are moved in the context of hypotheses. The number of *simple_intropattern* must not exceed the number of equalities newly generated. If it is smaller, fresh names are automatically generated to adjust the list of *simple_intropattern* to the number of new equalities. The original equality is erased if it corresponds to a hypothesis.

Variant: `injection` *term* `with` *bindings_list* ? `as` *injection_intropattern*
Variant: `injection` *num* `as` *injection_intropattern*
Variant: `injection as` *injection_intropattern*

Variant: `einjection` *term* `with` *bindings_list* ? `as` *injection_intropattern*
Variant: `einjection` *num* `as` *injection_intropattern*
Variant: `einjection as` *injection_intropattern*

These are equivalent to the previous variants but using instead the syntax *injection_intropattern* which *intros* uses. In particular `as [=` *simple_intropattern* + `]` behaves the same as `as` *simple_intropattern* + .

Flag: `Structural Injection`

This flag ensures that `injection` *term* erases the original hypothesis and leaves the generated equalities in the context rather than putting them as antecedents of the current goal, as if giving `injection` *term* `as` (with an empty list of names). This flag is off by default.

Flag: `Keep Proof Equalities`

By default, *injection* only creates new equalities between *term*s whose type is in sort `Type` or `Set`, thus implementing a special behavior for objects that are proofs of a statement in `Prop`. This flag controls this behavior.

inversion *ident*

Let the type of *ident* in the local context be `(I t)`, where `I` is a (co)inductive predicate. Then, `inversion` applied to *ident* derives for each possible constructor `c i` of `(I t)`, all the necessary conditions that should hold for the instance `(I t)` to be proved by `c i`.

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

Note: 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. See *Generation of inversion principles with Derive Inversion*.

Note: Part of the behavior of the `inversion` tactic is to generate equalities between expressions that appeared in the hypothesis that is being processed. By default, no equalities are generated if they relate two proofs (i.e. equalities between *term*s whose type is in sort `Prop`). This behavior can be turned off by using the *Keep Proof Equalities* setting.

Variant: inversion *num*

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

Variant: inversion_clear *ident*

This behaves as `inversion` and then erases *ident* from the context.

Variant: inversion *ident* **as** *or_and_intropattern_loc*

This generally behaves as inversion but using names in *or_and_intropattern_loc* for naming hypotheses. The *or_and_intropattern_loc* must have the form `[`p_{11} `...` p_{1n} `| ... |` p_{m1} `...` p_{mn} `]` 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 = 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} `...` p_{in} 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} `...` p_{ijq} `]` (or, equivalently, `(`p_{ij1} `, ...,` p_{ijq} `)`) where `q` is the number of subequalities obtained from splitting the original equation. Here is an example. The `inversion ... as` variant of `inversion` generally behaves in a slightly more expectable way than `inversion` (no artificial duplication of some hypotheses referring to other hypotheses). To take benefit of these improvements, it is enough to use `inversion ... as []`, letting the names being finally chosen by Coq.

Example

```
Inductive contains0 : list nat -> Prop :=
| in_hd : forall l, contains0 (0 :: l)
| in_tl : forall l b, contains0 l -> contains0 (b :: l).
    contains0 is defined
    contains0_ind is defined
    contains0_sind is defined
```

(continues on next page)

(continued from previous page)

```
Goal forall l:list nat, contains0 (1 :: l) -> contains0 l.
    1 subgoal

      ============================
      forall l : list nat, contains0 (1 :: l) -> contains0 l

intros l H; inversion H as [ | l' p Hl' [Heqp Heql'] ].
    1 subgoal

      l : list nat
      H : contains0 (1 :: l)
      l' : list nat
      p : nat
      Hl' : contains0 l
      Heqp : p = 1
      Heql' : l' = l
      ============================
      contains0 l
```

Variant: `inversion` *num* `as` *or_and_intropattern_loc*

This allows naming the hypotheses introduced by `inversion` *num* in the context.

Variant: `inversion_clear` *ident* `as` *or_and_intropattern_loc*

This allows naming the hypotheses introduced by `inversion_clear` in the context. Notice that hypothesis names can be provided as if `inversion` were called, even though the `inversion_clear` will eventually erase the hypotheses.

Variant: `inversion` *ident* `in` *ident*+

Let *ident*+ be identifiers in the local context. This tactic behaves as generalizing *ident*+, and then performing `inversion`.

Variant: `inversion` *ident* `as` *or_and_intropattern_loc* `in` *ident*+

This allows naming the hypotheses introduced in the context by `inversion` *ident* `in` *ident*+.

Variant: `inversion_clear` *ident* `in` *ident*+

Let *ident*+ be identifiers in the local context. This tactic behaves as generalizing *ident*+, and then performing `inversion_clear`.

Variant: `inversion_clear` *ident* `as` *or_and_intropattern_loc* `in` *ident*+

This allows naming the hypotheses introduced in the context by `inversion_clear` *ident* `in` *ident*+.

Variant: `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.

Variant: `dependent inversion` *ident* `as` *or_and_intropattern_loc*

This allows naming the hypotheses introduced in the context by `dependent inversion` *ident*.

Variant: `dependent inversion_clear` *ident*

Like `dependent inversion`, except that *ident* is cleared from the local context.

Variant: `dependent inversion_clear` *ident* `as` *or_and_intropattern_loc*

This allows naming the hypotheses introduced in the context by `dependent inversion_clear` *ident*.

Variant: `dependent inversion` *ident* `with` *term*

This variant allows you to specify the generalization of the goal. It is useful when the system fails to generalize the goal automatically. If *ident* has type `(I t)` and `I` has type `forall (x:T), s`, then *term* must be of type `I:forall (x:T), I x -> s'` where `s'` is the type of the goal.

Variant: `dependent inversion` *ident* `as` *or_and_intropattern_loc* `with` *term*

This allows naming the hypotheses introduced in the context by `dependent inversion` *ident* `with` *term*.

Variant: `dependent inversion_clear` *ident* `with` *term*

Like *dependent inversion ... with ...* with but clears *ident* from the local context.

Variant: `dependent inversion_clear` *ident* `as` *or_and_intropattern_loc* `with` *term*

This allows naming the hypotheses introduced in the context by `dependent inversion_clear` *ident* `with` *term*.

Variant: `simple inversion` *ident*

It is a very primitive inversion tactic that derives all the necessary equalities but it does not simplify the constraints as `inversion` does.

Variant: `simple inversion` *ident* `as` *or_and_intropattern_loc*

This allows naming the hypotheses introduced in the context by `simple inversion`.

Variant: `inversion` *ident* `using` *ident*

Let *ident* have type `(I 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.

Variant: `inversion` *ident* `using` *ident* `in` *ident*+

This tactic behaves as generalizing *ident*+, then doing `inversion` *ident* `using` *ident*.

Variant: `inversion_sigma`

This tactic turns equalities of dependent pairs (e.g., `existT P x p = existT P y q`, frequently left over by inversion on a dependent type family) into pairs of equalities (e.g., a hypothesis `H : x = y` and a hypothesis of type `rew H in p = q`); these hypotheses can subsequently be simplified using *subst*, without ever invoking any kind of axiom asserting uniqueness of identity proofs. If you want to explicitly specify the hypothesis to be inverted, or name the generated hypotheses, you can invoke `induction H as [H1 H2] using eq_sigT_rect`. This tactic also works for `sig`, `sigT2`, and `sig2`, and there are similar `eq_sig ***_rect` induction lemmas.

Example

Non-dependent inversion.

Let us consider the relation `Le` over natural numbers:

```
Inductive Le : nat -> nat -> Set :=
| LeO : forall n:nat, Le 0 n
| LeS : forall n m:nat, Le n m -> Le (S n) (S m).
```

Let us consider the following goal:

```
1 subgoal

  P : nat -> nat -> Prop
  Q : forall n m : nat, Le n m -> Prop
  n, 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 proving that the only possible constructor of (Le (S n) m) is LeS and that we can invert the arrow in the type of LeS. This inversion is possible because Le is the smallest set closed by the constructors LeO and LeS.

```
inversion_clear H.
    1 subgoal

      P : nat -> nat -> Prop
      Q : forall n m : nat, Le n m -> Prop
      n, m, 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:

```
inversion H.
    1 subgoal

      P : nat -> nat -> Prop
      Q : forall n m : nat, Le n m -> Prop
      n, m : nat
      H : Le (S n) m
      n0, m0 : nat
      H1 : Le n m0
      H0 : n0 = n
      H2 : S m0 = m
      ============================
      P n (S m0)
```

Example

Dependent inversion.

Let us consider the following goal:

```
    1 subgoal

      P : nat -> nat -> Prop
      Q : forall n m : nat, Le n m -> Prop
      n, 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* do such a substitution. To have such a behavior we use the dependent inversion tactics:

```
dependent inversion_clear H.
    1 subgoal
```

(continues on next page)

(continued from previous page)

```
      P : nat -> nat -> Prop
      Q : forall n m : nat, Le n m -> Prop
      n, m, 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

Using inversion_sigma.

Let us consider the following inductive type of length-indexed lists, and a lemma about inverting equality of cons:

```
Require Import Coq.Logic.Eqdep_dec.
Inductive vec A : nat -> Type :=
| nil : vec A O
| cons {n} (x : A) (xs : vec A n) : vec A (S n).
    vec is defined
    vec_rect is defined
    vec_ind is defined
    vec_rec is defined
    vec_sind is defined

Lemma invert_cons : forall A n x xs y ys,
         @cons A n x xs = @cons A n y ys
         -> xs = ys.
    1 subgoal

      ============================
      forall (A : Type) (n : nat) (x : A) (xs : vec A n) (y : A) (ys : vec A n),
      cons A x xs = cons A y ys -> xs = ys

Proof.
intros A n x xs y ys H.
    1 subgoal

      A : Type
      n : nat
      x : A
      xs : vec A n
      y : A
      ys : vec A n
      H : cons A x xs = cons A y ys
      ============================
      xs = ys
```

After performing inversion, we are left with an equality of existTs:

```
inversion H.
    1 subgoal
```

(continues on next page)

(continued from previous page)

```
      A : Type
      n : nat
      x : A
      xs : vec A n
      y : A
      ys : vec A n
      H : cons A x xs = cons A y ys
      H1 : x = y
      H2 : existT (fun n : nat => vec A n) n xs =
           existT (fun n : nat => vec A n) n ys
      ============================
      xs = ys
```

We can turn this equality into a usable form with inversion_sigma:

```
inversion_sigma.
    1 subgoal

      A : Type
      n : nat
      x : A
      xs : vec A n
      y : A
      ys : vec A n
      H : cons A x xs = cons A y ys
      H1 : x = y
      H0 : n = n
      H3 : eq_rect n (fun a : nat => vec A a) xs n H0 = ys
      ============================
      xs = ys
```

To finish cleaning up the proof, we will need to use the fact that that all proofs of n = n for n a nat are eq_refl:

```
let H := match goal with H : n = n |- _ => H end in
pose proof (Eqdep_dec.UIP_refl_nat _ H); subst H.
    1 subgoal

      A : Type
      n : nat
      x : A
      xs : vec A n
      y : A
      ys : vec A n
      H : cons A x xs = cons A y ys
      H1 : x = y
      H3 : eq_rect n (fun a : nat => vec A a) xs n eq_refl = ys
      ============================
      xs = ys

simpl in *.
    1 subgoal

      A : Type
      n : nat
      x : A
      xs : vec A n
```

(continues on next page)

(continued from previous page)

```
      y : A
      ys : vec A n
      H : cons A x xs = cons A y ys
      H1 : x = y
      H3 : xs = ys
      ============================
      xs = ys
```

Finally, we can finish the proof:

```
assumption.
    No more subgoals.

Qed.
```

fix *ident num*

This tactic is a primitive tactic to start a proof by induction. In general, it is easier to rely on higher-level induction tactics such as the ones described in *induction*.

In the syntax of the tactic, the identifier *ident* is the name given to the induction hypothesis. The natural number *num* tells on which premise of the current goal the induction acts, starting from 1, counting both dependent and non dependent products, but skipping local definitions. Especially, the current lemma must be composed of at least *num* products.

Like in a fix expression, the induction hypotheses have to be used on structurally smaller arguments. The verification that inductive proof arguments are correct is done only at the time of registering the lemma in the environment. To know if the use of induction hypotheses is correct at some time of the interactive development of a proof, use the command `Guarded` (see Section *Requesting information*).

Variant: `fix` *ident num* `with` (*ident* *binder*+ [{`struct` *ident*}] : *type*)+

This starts a proof by mutual induction. The statements to be simultaneously proved are respectively `forall binder ... binder, type`. The identifiers *ident* are the names of the induction hypotheses. The identifiers *ident* are the respective names of the premises on which the induction is performed in the statements to be simultaneously proved (if not given, the system tries to guess itself what they are).

cofix *ident*

This tactic starts a proof by coinduction. The identifier *ident* is the name given to the coinduction hypothesis. Like in a cofix expression, the use of induction hypotheses have to guarded by a constructor. The verification that the use of co-inductive hypotheses is correct is done only at the time of registering the lemma in the environment. To know if the use of coinduction hypotheses is correct at some time of the interactive development of a proof, use the command `Guarded` (see Section *Requesting information*).

Variant: `cofix` *ident* `with` (*ident* *binder*+ : *type*)+

This starts a proof by mutual coinduction. The statements to be simultaneously proved are respectively `forall binder ... binder, type` The identifiers *ident* are the names of the coinduction hypotheses.

5.3.6 Rewriting expressions

These tactics use the equality `eq:forall A:Type, A->A->Prop` defined in file `Logic.v` (see *Logic*). The notation for `eq T t u` is simply `t=u` dropping the implicit type of `t` and `u`.

rewrite *term*

This tactic applies to any goal. The type of *term* must have the form

`forall` $(x_1 : A_1) \ldots (x_n : A_n)$, `eq` $term_1$ $term_2$.

where `eq` is the Leibniz equality or a registered setoid equality.

Then `rewrite` *term* finds the first subterm matching $term_1$ in the goal, resulting in instances $term_1$' and $term_2$' and then replaces every occurrence of $term_1$' by $term_2$'. Hence, some of the variables x_i are solved by unification, and some of the types $A_1, \ldots, A_n$ become new subgoals.

Error: `The` *term* `provided does not end with an equation.`

Error: `Tactic generated a subgoal identical to the original goal. This happens if` *term* `does not occ`

Variant: `rewrite ->` *term*

Is equivalent to `rewrite` *term*

Variant: `rewrite <-` *term*

Uses the equality $term_1$ = $term_2$ from right to left

Variant: `rewrite` *term* `in` *goal_occurrences*

Analogous to `rewrite` *term* but rewriting is done following the clause *goal_occurrences*. For instance:

- `rewrite H in H'` will rewrite `H` in the hypothesis `H'` instead of the current goal.
- `rewrite H in H' at 1, H'' at - 2 |- *` means `rewrite H; rewrite H in H' at 1; rewrite H in H'' at - 2`. In particular a failure will happen if any of these three simpler tactics fails.
- `rewrite H in * |-` will do `rewrite H in H'` for all hypotheses `H'` different from `H`. A success will happen as soon as at least one of these simpler tactics succeeds.
- `rewrite H in *` is a combination of `rewrite H` and `rewrite H in * |-` that succeeds if at least one of these two tactics succeeds.

Orientation `->` or `<-` can be inserted before the *term* to rewrite.

Variant: `rewrite` *term* `at` *occurrences*

Rewrite only the given *occurrences* of *term*. Occurrences are specified from left to right as for pattern (*pattern*). The rewrite is always performed using setoid rewriting, even for Leibniz's equality, so one has to `Import Setoid` to use this variant.

Variant: `rewrite` *term* `by` *tactic*

Use tactic to completely solve the side-conditions arising from the *rewrite*.

Variant: `rewrite` [*orientation term*]$^{+}_{,}$ [`in` *ident*]$^{?}$

Is equivalent to the `n` successive tactics [`rewrite` *term*]$^{+}_{;}$, each one working on the first subgoal generated by the previous one. An `orientation ->` or `<-` can be inserted before each *term* to rewrite. One unique clause can be added at the end after the keyword in; it will then affect all rewrite operations.

In all forms of rewrite described above, a *term* to rewrite can be immediately prefixed by one of the following modifiers:

- `?` : the tactic `rewrite ?`*term* performs the rewrite of *term* as many times as possible (perhaps zero time). This form never fails.
- *num*`?` : works similarly, except that it will do at most *num* rewrites.
- `!` : works as `?`, except that at least one rewrite should succeed, otherwise the tactic fails.

- *num*! (or simply *num*) : precisely *num* rewrites of *term* will be done, leading to failure if these *num* rewrites are not possible.

Variant: `erewrite` *term*

This tactic works as `rewrite` *term* but turning unresolved bindings into existential variables, if any, instead of failing. It has the same variants as *rewrite* has.

Flag: `Keyed Unification`

Makes higher-order unification used by *rewrite* rely on a set of keys to drive unification. The subterms, considered as rewriting candidates, must start with the same key as the left- or right-hand side of the lemma given to rewrite, and the arguments are then unified up to full reduction.

`replace` *term* `with` *term*'

This tactic applies to any goal. It replaces all free occurrences of *term* in the current goal with *term*' and generates an equality *term* = *term*' as a subgoal. This equality is automatically solved if it occurs among the assumptions, or if its symmetric form occurs. It is equivalent to `cut` *term* `=` *term*'`;` `[intro` H_n `; rewrite <-` H_n `; clear` H_n`|| assumption || symmetry; try assumption]`.

Error: `Terms do not have convertible types.`

Variant: `replace` *term* `with` *term*' `by` *tactic*

This acts as `replace` *term* `with` *term*' but applies `tactic` to solve the generated subgoal *term* `=` *term*'.

Variant: `replace` *term*

Replaces *term* with *term*' using the first assumption whose type has the form *term* `=` *term*' or *term*' `=` *term*.

Variant: `replace ->` *term*

Replaces *term* with *term*' using the first assumption whose type has the form *term* `=` *term*'

Variant: `replace <-` *term*

Replaces *term* with *term*' using the first assumption whose type has the form *term*' `=` *term*

Variant: `replace` *term* [`with` *term*]? `in` *goal_occurrences* [`by` *tactic*]?

Variant: `replace ->` *term* `in` *goal_occurrences*

Variant: `replace <-` *term* `in` *goal_occurrences*

Acts as before but the replacements take place in the specified clauses (*goal_occurrences*) (see *Performing computations*) and not only in the conclusion of the goal. The clause argument must not contain any `type of` nor `value of`.

Variant: `cutrewrite <- (`*term* `=` *term*'`)`

Deprecated since version 8.5: This tactic can be replaced by `enough (`*term* `=` *term*'`) as <-`.

Variant: `cutrewrite -> (`*term* `=` *term*'`)`

Deprecated since version 8.5: This tactic can be replaced by `enough (`*term* `=` *term*'`) as ->`.

`subst` *ident*

This tactic applies to a goal that has *ident* in its context and (at least) one hypothesis, say `H`, of type *ident* `= t` or `t =` *ident* with *ident* not occurring in `t`. 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.

If *ident* is a local definition of the form *ident* `:= t`, it is also unfolded and cleared.

Note:

- When several hypotheses have the form *ident* `= t` or `t =` *ident*, the first one is used.
- If `H` is itself dependent in the goal, it is replaced by the proof of reflexivity of equality.

Variant: `subst` *ident*$^+$

This is equivalent to `subst` *ident*$_1$`; ...; subst` *ident*$_n$.

Variant: `subst`

This applies subst repeatedly from top to bottom to all identifiers of the context for which an equality of the form *ident* `= t` or `t =` *ident* or *ident* `:= t` exists, with *ident* not occurring in `t`.

Flag: `Regular Subst Tactic`

This flag controls the behavior of *subst*. When it is activated (it is by default), *subst* also deals with the following corner cases:

- A context with ordered hypotheses *ident*$_1$ `=` *ident*$_2$ and *ident*$_1$ `= t`, or `t =` *ident*$_1$' with `t` not a variable, and no other hypotheses of the form *ident*$_2$ `= u` or `u =` *ident*$_2$; without the flag, a second call to subst would be necessary to replace *ident*$_2$ by `t` or `t` respectively.
- The presence of a recursive equation which without the flag would be a cause of failure of *subst*.
- A context with cyclic dependencies as with hypotheses *ident*$_1$ `= f` *ident*$_2$ and *ident*$_2$ `= g` *ident*$_1$ which without the flag would be a cause of failure of *subst*.

Additionally, it prevents a local definition such as *ident* `:= t` to be unfolded which otherwise it would exceptionally unfold in configurations containing hypotheses of the form *ident* `= u`, or `u =` *ident* with `u` not a variable. Finally, it preserves the initial order of hypotheses, which without the flag it may break. default.

`stepl` *term*

This tactic is for chaining rewriting steps. It assumes a goal of the form `R` *term term* where `R` is a binary relation and relies on a database of lemmas of the form `forall x y z, R x y -> eq x z -> R z y` where `eq` is typically a setoid equality. The application of `stepl` *term* then replaces the goal by `R` *term term* and adds a new goal stating `eq` *term term*.

Command: `Declare Left Step` *term*

Adds *term* to the database used by *stepl*.

This tactic is especially useful for parametric setoids which are not accepted as regular setoids for *rewrite* and *setoid_replace* (see *Generalized rewriting*).

Variant: `stepl` *term* `by` *tactic*

This applies `stepl` *term* then applies `tactic` to the second goal.

Variant: `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`.

Command: `Declare Right Step` *term*

Adds *term* to the database used by *stepr*.

`change` *term*

This tactic applies to any goal. It implements the rule `Conv` given in *Subtyping rules*. `change U` replaces the current goal `T` with `U` providing that `U` is well-formed and that `T` and `U` are convertible.

Error: `Not convertible.`

Variant: `change` *term* `with` *term*'

This replaces the occurrences of *term* by *term*' in the current goal. The term *term* and *term*' must be convertible.

Variant: `change` *term* `at` *num*$^+$ `with` *term*'

This replaces the occurrences numbered *num*$^+$ of *term* by *term*' in the current goal. The terms *term* and *term*' must be convertible.

Error: `Too few occurrences.`

Variant: `change` *term* `at` *num*+ ? `with` *term* ? `in` *ident*

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

Variant: `now_show` *term*

This is a synonym of `change` *term*. It can be used to make some proof steps explicit when refactoring a proof script to make it readable.

See also:

Performing computations

5.3.7 Performing computations

This set of tactics implements different specialized usages of the tactic *change*.

All conversion tactics (including *change*) can be parameterized by the parts of the goal where the conversion can occur. This is done using *goal clauses* which consists in a list of hypotheses and, optionally, of a reference to the conclusion of the goal. For defined hypothesis it is possible to specify if the conversion should occur on the type part, the body part or both (default).

Goal clauses are written after a conversion tactic (tactics *set*, *rewrite*, *replace* and *autorewrite* also use goal clauses) and are introduced by the keyword `in`. If no goal clause is provided, the default is to perform the conversion only in the conclusion.

The syntax and description of the various goal clauses is the following:

- `in` *ident*+ `|-` only in hypotheses *ident*+
- `in` *ident*+ `|- *` in hypotheses *ident*+ and in the conclusion
- `in * |-` in every hypothesis
- `in *` (equivalent to in `* |- *`) everywhere
- `in (type of` *ident*`) (value of` *ident*`) ... |-` in type part of *ident*, in the value part of *ident*, etc.

For backward compatibility, the notation `in` *ident*+ performs the conversion in hypotheses *ident*+.

`cbv` *flag**

`lazy` *flag**

These parameterized reduction tactics apply to any goal and perform the normalization of the goal according to the specified flags. In correspondence with the kinds of reduction considered in Coq namely β (reduction of functional application), δ (unfolding of transparent constants, see *Controlling the reduction strategies and the conversion algorithm*), ι (reduction of pattern matching over a constructed term, and unfolding of `fix` and `cofix` expressions) and ζ (contraction of local definitions), the flags are either `beta`, `delta`, `match`, `fix`, `cofix`, `iota` or `zeta`. The `iota` flag is a shorthand for `match`, `fix` and `cofix`. The `delta` flag itself can be refined into `delta [` *qualid*+ `]` or `delta - [` *qualid*+ `]`, restricting in the first case the constants to unfold to the constants listed, and restricting in the second case the constant to unfold to all but the ones explicitly mentioned. Notice that the `delta` flag does not apply to variables bound by a let-in construction inside the *term* itself (use here the `zeta` flag). In any cases, opaque constants are not unfolded (see *Controlling the reduction strategies and the conversion algorithm*).

Normalization according to the flags is done by first evaluating the head of the expression into a *weak-head* normal form, i.e. until the evaluation is blocked by a variable (or an opaque constant, or an axiom), as e.g. in `x u1 ... un` , or `match x with ... end`, or `(fix f x {struct x} := ...) x`, or is a constructed form (a λ-expression, a constructor, a cofixpoint, an inductive type, a product type, a sort), or is a redex that the flags prevent to reduce. Once a weak-head normal form is obtained, subterms are recursively reduced using the same strategy.

Reduction to weak-head normal form can be done using 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 weakly evaluated only when necessary, and if an argument is used several times then it is weakly computed only once. This reduction is efficient for reducing expressions with dead code. For instance, the proofs of a proposition `exists x. P(x)` reduce to a pair of a witness `t`, and a proof that `t` satisfies the predicate `P`. Most of the time, `t` may be computed without computing the proof of `P(t)`, thanks to the lazy strategy.

The call-by-value strategy is the one used in ML languages: the arguments of a function call are systematically weakly evaluated first. Despite the lazy strategy always performs fewer reductions than the call-by-value strategy, the latter is generally more efficient for evaluating purely computational expressions (i.e. with little dead code).

Variant: `compute`
Variant: `cbv`

These are synonyms for `cbv beta delta iota zeta`.

Variant: `lazy`

This is a synonym for `lazy beta delta iota zeta`.

Variant: `compute [` *qualid*+ `]`

Variant: `cbv [` *qualid*+ `]`

These are synonyms of `cbv beta delta` *qualid*+ `iota zeta`.

Variant: `compute - [` *qualid*+ `]`

Variant: `cbv - [` *qualid*+ `]`

These are synonyms of `cbv beta delta -`*qualid*+ `iota zeta`.

Variant: `lazy [` *qualid*+ `]`

Variant: `lazy - [` *qualid*+ `]`

These are respectively synonyms of `lazy beta delta` *qualid*+ `iota zeta` and `lazy beta delta -`*qualid*+ `iota zeta`.

Variant: `vm_compute`

This tactic evaluates the goal using the optimized call-by-value evaluation bytecode-based virtual machine described in *[GregoireL02]*. This algorithm is dramatically more efficient than the algorithm used for the *cbv* tactic, but it cannot be fine-tuned. It is especially interesting for full evaluation of algebraic objects. This includes the case of reflection-based tactics.

Variant: `native_compute`

This tactic evaluates the goal by compilation to OCaml as described in *[BDenesGregoire11]*. If Coq is running in native code, it can be typically two to five times faster than *vm_compute*. Note however that the compilation cost is higher, so it is worth using only for intensive computations.

Flag: NativeCompute Profiling

On Linux, if you have the `perf` profiler installed, this flag makes it possible to profile *native_compute* evaluations.

Option: NativeCompute Profile Filename *string*

This option specifies the profile output; the default is `native_compute_profile.data`. The actual filename used will contain extra characters to avoid overwriting an existing file; that filename is reported to the user. That means you can individually profile multiple uses of *native_compute* in a script. From the Linux command line, run `perf report` on the profile file to see the results. Consult the `perf` documentation for more details.

Flag: Debug Cbv

This flag makes *cbv* (and its derivative *compute*) print information about the constants it encounters and the unfolding decisions it makes.

red

This tactic applies to a goal that has the form:

```
forall (x:T1) ... (xk:Tk), T
```

with `T` $\beta\iota\zeta$-reducing to `c` t_1 ... t_n and `c` a constant. If `c` is transparent then it replaces `c` with its definition (say `t`) and then reduces (`t` t_1 ... t_n) according to $\beta\iota\zeta$-reduction rules.

Error: Not reducible.

Error: No head constant to reduce.

hnf

This tactic applies to any goal. It replaces the current goal with its head normal form according to the $\beta\delta\iota\zeta$-reduction rules, i.e. it reduces the head of the goal until it becomes a product or an irreducible term. All inner $\beta\iota$-redexes are also reduced.

Example: The term `fun n : nat => S n + S n` is not reduced by `hnf`.

Note: The δ rule only applies to transparent constants (see *Controlling the reduction strategies and the conversion algorithm* on transparency and opacity).

cbn
simpl

These tactics apply to any goal. They try to reduce a term to something still readable instead of fully normalizing it. They perform a sort of strong normalization with two key differences:

- They unfold a constant if and only if it leads to a ι-reduction, i.e. reducing a match or unfolding a fixpoint.
- While reducing a constant unfolding to (co)fixpoints, the tactics use the name of the constant the (co)fixpoint comes from instead of the (co)fixpoint definition in recursive calls.

The *cbn* tactic is claimed to be a more principled, faster and more predictable replacement for *simpl*.

The *cbn* tactic accepts the same flags as *cbv* and *lazy*. The behavior of both *simpl* and *cbn* can be tuned using the Arguments vernacular command as follows:

- A constant can be marked to be never unfolded by *cbn* or *simpl*:

Example

```
Arguments minus n m : simpl never.
```

After that command an expression like (minus (S x) y) is left untouched by the tactics *cbn* and *simpl*.

- A constant can be marked to be unfolded only if applied to enough arguments. The number of arguments required can be specified using the / symbol in the argument list of the *Arguments* vernacular command.

Example

```
Definition fcomp A B C f (g : A -> B) (x : A) : C := f (g x).
    fcomp is defined

Arguments fcomp {A B C} f g x /.
Notation "f \o g" := (fcomp f g) (at level 50).
```

After that command the expression (f \o g) is left untouched by *simpl* while ((f \o g) t) is reduced to (f (g t)). The same mechanism can be used to make a constant volatile, i.e. always unfolded.

Example

```
Definition volatile := fun x : nat => x.
    volatile is defined

Arguments volatile / x.
```

- A constant can be marked to be unfolded only if an entire set of arguments evaluates to a constructor. The ! symbol can be used to mark such arguments.

Example

```
Arguments minus !n !m.
```

After that command, the expression (minus (S x) y) is left untouched by *simpl*, while (minus (S x) (S y)) is reduced to (minus x y).

- A special heuristic to determine if a constant has to be unfolded can be activated with the following command:

Example

```
Arguments minus n m : simpl nomatch.
```

The heuristic avoids to perform a simplification step that would expose a match construct in head position. For example the expression (minus (S (S x)) (S y)) is simplified to (minus (S x) y) even if an extra simplification is possible.

In detail, the tactic *simpl* first applies $\beta\iota$-reduction. Then, it expands transparent constants and tries to reduce further using $\beta\iota$-reduction. But, when no ι rule is applied after unfolding then δ-reductions are not applied. For instance trying to use *simpl* on (plus n O) = n changes nothing.

Notice that only transparent constants whose name can be reused in the recursive calls are possibly unfolded by *simpl*. For instance a constant defined by `plus' := plus` is possibly unfolded and reused in the recursive calls, but a constant such as `succ := plus (S O)` is never unfolded. This is the main difference between *simpl* and *cbn*. The tactic *cbn* reduces whenever it will be able to reuse it or not: `succ t` is reduced to `S t`.

Variant: `cbn [` *qualid*+ `]`

Variant: `cbn - [` *qualid*+ `]`

These are respectively synonyms of `cbn beta delta [` *qualid*+ `] iota zeta` and `cbn beta delta - [` *qualid*+ `] iota zeta` (see *cbn*).

Variant: `simpl` *pattern*

This applies *simpl* only to the subterms matching *pattern* in the current goal.

Variant: `simpl` *pattern* `at` *num*+

This applies *simpl* only to the *num*+ occurrences of the subterms matching *pattern* in the current goal.

Error: `Too few occurrences.`

Variant: `simpl` *qualid*
Variant: `simpl` *string*

This applies *simpl* only to the applicative subterms whose head occurrence is the unfoldable constant *qualid* (the constant can be referred to by its notation using *string* if such a notation exists).

Variant: `simpl` *qualid* `at` *num*+
Variant: `simpl` *string* `at` *num*+

This applies *simpl* only to the *num*+ applicative subterms whose head occurrence is *qualid* (or *string*).

Flag: `Debug RAKAM`

This flag makes *cbn* print various debugging information. `RAKAM` is the Refolding Algebraic Krivine Abstract Machine.

`unfold` *qualid*

This tactic applies to any goal. The argument qualid must denote a defined transparent constant or local definition (see *Definitions* and *Controlling the reduction strategies and the conversion algorithm*). 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 $\beta\iota\zeta$-normal form. Use the general reduction tactics if you want to avoid this final reduction, for instance `cbv delta [`*qualid*`]`.

Error: `Cannot coerce` *qualid* `to an evaluable reference.`

This error is frequent when trying to unfold something that has defined as an inductive type (or constructor) and not as a definition.

Example

```
Goal 0 <= 1.
    1 subgoal

      ============================
      0 <= 1

unfold le.
```

(continues on next page)

(continued from previous page)

```
    Toplevel input, characters 0-10:
    > unfold le.
    > ^^^^^^^^^^
    Error: Cannot coerce le to an evaluable reference.
```

This error can also be raised if you are trying to unfold something that has been marked as opaque.

Example

```
Opaque Nat.add.
Goal 1 + 0 = 1.
    1 subgoal

      ============================
      1 + 0 = 1

unfold Nat.add.
    Toplevel input, characters 0-15:
    > unfold Nat.add.
    > ^^^^^^^^^^^^^^^
    Error: Cannot coerce Nat.add to an evaluable reference.
```

Variant: `unfold` *qualid* `in` *goal_occurrences*

Replaces *qualid* in hypothesis (or hypotheses) designated by *goal_occurrences* with its definition and replaces the hypothesis with its $\beta\iota$ normal form.

Variant: `unfold` [*qualid*]+,

Replaces [*qualid*]+, with their definitions and replaces the current goal with its $\beta\iota$ normal form.

Variant: `unfold` [*qualid* `at` *occurrences*]+,

The list *occurrences* specify the occurrences of *qualid* to be unfolded. Occurrences are located from left to right.

Error: `Bad occurrence number of` *qualid*.

Error: *qualid* `does not occur.`

Variant: `unfold` *string*

If *string* denotes the discriminating symbol of a notation (e.g. "+") or an expression defining a notation (e.g. `"_ + _"`), and this notation denotes an application whose head symbol is an unfoldable constant, then the tactic unfolds it.

Variant: `unfold` *string*`%`*ident*

This is variant of `unfold` *string* where *string* gets its interpretation from the scope bound to the delimiting key *ident* instead of its default interpretation (see *Local interpretation rules for notations*).

Variant: `unfold` [[*qualid* | *string*[`%`*ident*]?] [`at` *occurrences*]?]+, [`in` *goal_occurrences*]?

This is the most general form.

`fold` *term*

This tactic applies to any goal. The term *term* is reduced using the *red* tactic. Every occurrence of the resulting *term* in the goal is then replaced by *term*. This tactic is particularly useful when a fixpoint definition has been wrongfully unfolded, making the goal very hard to read. On the other hand, when an unfolded function applied to its argument has been reduced, the *fold* tactic won't do anything.

Example

```
Goal ~0=0.
    1 subgoal

      ============================
      0 <> 0

unfold not.
    1 subgoal

      ============================
      0 = 0 -> False

Fail progress fold not.
    The command has indeed failed with message:
    Failed to progress.

pattern (0 = 0).
    1 subgoal

      ============================
      (fun P : Prop => P -> False) (0 = 0)

fold not.
    1 subgoal

      ============================
      0 <> 0
```

Variant: `fold` *term*+

Equivalent to `fold` *term* `; ... ; fold` *term*.

`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

- replacing all occurrences of *term* in `T` with a fresh variable
- abstracting this variable
- applying the abstracted goal to *term*

For instance, if the current goal `T` is expressible as φ`(t)` where the notation captures all the instances of `t` in φ`(t)`, then `pattern t` transforms it into `(fun x:A =>` φ`(x)) t`. This tactic can be used, for instance, when the tactic `apply` fails on matching.

Variant: `pattern` *term* `at` *num*+

Only the occurrences *num*+ of *term* are considered for β-expansion. Occurrences are located from left to right.

Variant: `pattern` *term* `at - ` *num*+

All occurrences except the occurrences of indexes *num*+ of *term* are considered for β-expansion. Occurrences are located from left to right.

Variant: `pattern` *term*+,

Starting from a goal φ(t$_1$ ... t$_m$), the tactic `pattern` t$_1$, ..., t$_m$ generates the equivalent goal (fun (x$_1$:A$_1$) ... (x$_m$:A$_m$) =>φ(x$_1$ ... x$_m$)) t$_1$ ... t$_m$. If t$_i$ occurs in one of the generated types A$_j$ these occurrences will also be considered and possibly abstracted.

Variant: `pattern` (*term* `at` *num*+)+,

This behaves as above but processing only the occurrences *num*+ of *term* starting from *term*.

Variant: `pattern` (*term* (`at` `-`? (*num*+,))?)+,

This is the most general syntax that combines the different variants.

Conversion tactics applied to hypotheses

tactic `in` *ident*+,

Applies `tactic` (any of the conversion tactics listed in this section) to the hypotheses *ident*+.

If *ident* is a local definition, then *ident* can be replaced by `type of` *ident* to address not the body but the type of the local definition.

Example: `unfold not in (type of H1) (type of H3)`.

Error: `No such hypothesis:` *ident*.

5.3.8 Automation

`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 introduces 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`.

> **Warning:** *auto* uses a weaker version of *apply* that is closer to *simple apply* so it is expected that sometimes *auto* will fail even if applying manually one of the hints would succeed.

Variant: `auto` *num*

Forces the search depth to be *num*. The maximal search depth is 5 by default.

Variant: `auto with` *ident*+

Uses the hint databases *ident*+ in addition to the database `core`.

Note: Use the fake database `nocore` if you want to *not* use the `core` database.

Variant: `auto with *`

Uses all existing hint databases. Using this variant is highly discouraged in finished scripts since it is both slower and less robust than the variant where the required databases are explicitly listed.

See also:

The Hints Databases for auto and eauto for the list of pre-defined databases and the way to create or extend a database.

Variant: `auto using` *qualid*$_i$`+` `with` *ident*`+` `?`

Uses lemmas *qualid*$_i$ in addition to hints. If *qualid* is an inductive type, it is the collection of its constructors which are added as hints.

Note: The hints passed through the `using` clause are used in the same way as if they were passed through a hint database. Consequently, they use a weaker version of *apply* and `auto using` *qualid* may fail where `apply` *qualid* succeeds.

Given that this can be seen as counter-intuitive, it could be useful to have an option to use full-blown *apply* for lemmas passed through the `using` clause. Contributions welcome!

Variant: `info_auto`

Behaves like *auto* but shows the tactics it uses to solve the goal. This variant is very useful for getting a better understanding of automation, or to know what lemmas/assumptions were used.

Variant: `debug auto`

Behaves like *auto* but shows the tactics it tries to solve the goal, including failing paths.

Variant: `info_` `?` `auto` *num* `?` `using` *qualid*`+` `?` `with` *ident*`+` `?`

This is the most general form, combining the various options.

Variant: `trivial`

This tactic is a restriction of *auto* that is not recursive and tries only hints that cost `0`. Typically it solves trivial equalities like `X=X`.

Variant: `trivial with` *ident*`+`

Variant: `trivial with *`

Variant: `trivial using` *qualid*`+`

Variant: `debug trivial`

Variant: `info_trivial`

Variant: `info_` `?` `trivial` `using` *qualid*`+` `?` `with` *ident*`+` `?`

Note: *auto* and *trivial* either solve completely the goal or else succeed without changing the goal. Use `solve [ auto ]` and `solve [ trivial ]` if you would prefer these tactics to fail when they do not manage to solve the goal.

Flag: `Info Auto`

Flag: `Debug Auto`

Flag: `Info Trivial`

Flag: Debug Trivial

These flags enable printing of informative or debug information for the *auto* and *trivial* tactics.

eauto

This tactic generalizes *auto*. While *auto* does not try resolution hints which would leave existential variables in the goal, *eauto* does try them (informally speaking, it internally uses a tactic close to *simple eapply* instead of a tactic close to *simple apply* in the case of *auto*). As a consequence, *eauto* can solve such a goal:

Example

```
Hint Resolve ex_intro : core.
    The hint ex_intro will only be used by eauto, because applying ex_intro would
    leave variable x as unresolved existential variable.

Goal forall P:nat -> Prop, P 0 -> exists n, P n.
    1 subgoal

      ============================
      forall P : nat -> Prop, P 0 -> exists n : nat, P n

eauto.
    No more subgoals.
```

Note that ex_intro should be declared as a hint.

Variant: info_? eauto *num*? using *qualid*+? with *ident*+?

The various options for *eauto* are the same as for *auto*.

eauto also obeys the following flags:

Flag: Info Eauto
Flag: Debug Eauto

See also:

The Hints Databases for auto and eauto

autounfold with *ident*+

This tactic unfolds constants that were declared through a *Hint Unfold* in the given databases.

Variant: autounfold with *ident*+ **in** *goal_occurrences*

Performs the unfolding in the given clause (*goal_occurrences*).

Variant: autounfold with *

Uses the unfold hints declared in all the hint databases.

autorewrite with *ident*+

This tactic carries out rewritings according to the rewriting rule bases *ident*+.

Each rewriting rule from the base *ident* 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* command.

Warning: This tactic may loop if you build non terminating rewriting systems.

Variant: `autorewrite with` *ident*+ `using` *tactic*

Performs, in the same way, all the rewritings of the bases *ident*+ applying tactic to the main subgoal after each rewriting step.

Variant: `autorewrite with` *ident*+ `in` *qualid*

Performs all the rewritings in hypothesis *qualid*.

Variant: `autorewrite with` *ident*+ `in` *qualid* `using` *tactic*

Performs all the rewritings in hypothesis *qualid* applying *tactic* to the main subgoal after each rewriting step.

Variant: `autorewrite with` *ident*+ `in` *goal_occurrences*

Performs all the rewriting in the clause *goal_occurrences*.

See also:

Hint-Rewrite for feeding the database of lemmas used by *autorewrite* and *autorewrite* for examples showing the use of this tactic.

`easy`

This tactic tries to solve the current goal by a number of standard closing steps. In particular, it tries to close the current goal using the closing tactics *trivial*, *reflexivity*, *symmetry*, *contradiction* and *inversion* of hypothesis. If this fails, it tries introducing variables and splitting and-hypotheses, using the closing tactics afterwards, and splitting the goal using *split* and recursing.

This tactic solves goals that belong to many common classes; in particular, many cases of unsatisfiable hypotheses, and simple equality goals are usually solved by this tactic.

Variant: `now` *tactic*

Run *tactic* followed by *easy*. This is a notation for *tactic*`; easy`.

5.3.9 Controlling automation

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.

Command: `Print Hint` *ident*

Use this command to display the hints associated to the head symbol *ident* (see *Print Hint*). Each hint has a cost that is a nonnegative integer, and an optional pattern. The hints with lower cost are tried first. A hint is tried by *auto* when the conclusion of the current goal matches its pattern or when it has no pattern.

Creating Hint databases

One can optionally declare a hint database using the command *Create HintDb*. If a hint is added to an unknown database, it will be automatically created.

Command: `Create HintDb` *ident* `discriminated`?

This command creates a new database named *ident*. The database is implemented by a Discrimination Tree (DT) that serves as an index of all the lemmas. The DT can use transparency information to decide if a constant should be indexed or not (c.f. *The hints databases for auto and eauto*), making the retrieval more efficient. The legacy implementation (the default one for new databases) uses the DT only on goals without existentials (i.e., *auto* goals), for non-Immediate hints and does not make use of transparency hints, putting more work on the unification that is run after retrieval (it keeps a list of the lemmas in case the DT is not used). The new implementation enabled by the discriminated option makes use of DTs in all cases and takes transparency information into account. However, the order in which hints are retrieved from the DT may differ from the order in which they were inserted, making this implementation observationally different from the legacy one.

Command: `Hint` *hint_definition* `:` *ident*+

The general command to add a hint to some databases *ident*+. The various possible `hint_definition`s are given below.

Variant: `Hint` *hint_definition*

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

Deprecated since version 8.10.

Variant: `Local Hint` *hint_definition* `:` *ident*+

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 flag Local is useless since hints do not survive anyway to the closure of sections.

Variant: `Hint Resolve` *term* (`|` *num*? *pattern*?)? `:` *ident*

This command adds `simple 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 `simple apply` *term* or *num* if specified. The associated *pattern* is inferred from the conclusion of the type of *term* or the given *pattern* if specified. 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 however be reduced to a type starting with a product, the tactic `simple apply` *term* is also stored in the hints list. If the inferred type of *term* contains a dependent quantification on a variable which occurs only in the premisses of the type and not in its conclusion, no instance could be inferred for the variable by unification with the goal. In this case, the hint is added to the hint list of *eauto* instead of the hint list of auto and a warning is printed. A typical example of a hint that is used only by *eauto* is a transitivity lemma.

Error: *term* `cannot be used as a hint`

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

Variant: `Hint Resolve` *term*+ `:` *ident*

Adds each `Hint Resolve` *term*.

Variant: `Hint Resolve ->` *term* `:` *ident*

Adds the left-to-right implication of an equivalence as a hint (informally the hint will be used as `apply <-` *term*, although as mentioned before, the tactic actually used is a restricted version of *apply*).

Variant: `Hint Resolve <-` *term*

Adds the right-to-left implication of an equivalence as a hint.

Variant: `Hint Immediate` *term* `:` *ident*

This command adds `simple 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 `simple apply` *term* are not solved immediately by the *trivial* tactic (which only tries tactics with cost 0).This command is useful for theorems such as the symmetry of equality or `n+1=m+1 ->`

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: `term cannot be used as a hint`

Variant: `Hint Immediate term+ : ident`

Adds each `Hint Immediate term`.

Variant: `Hint Constructors qualid : ident`

If *qualid* 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 `(qualid ...)`, *auto* will try to apply each constructor.

Error: `qualid is not an inductive type`

Variant: `Hint Constructors qualid+ : ident`

Extends the previous command for several inductive types.

Variant: `Hint Unfold qualid : ident`

This adds the tactic `unfold qualid` to the hint list that will only be used when the head constant of the goal is *qualid*. Its cost is 4.

Variant: `Hint Unfold qualid+`

Extends the previous command for several defined constants.

Variant: `Hint Transparent qualid+ : ident`

Variant: `Hint Opaque qualid+ : ident`

This adds transparency hints to the database, making *qualid* transparent or opaque constants during resolution. This information is used during unification of the goal with any lemma in the database and inside the discrimination network to relax or constrain it in the case of discriminated databases.

Variant: `Hint Variables Transparent | Opaque : ident`

Variant: `Hint Constants Transparent | Opaque : ident`

This sets the transparency flag used during unification of hints in the database for all constants or all variables, overwriting the existing settings of opacity. It is advised to use this just after a *Create HintDb* command.

Variant: `Hint Extern num pattern? => tactic : ident`

This hint type is to extend *auto* with tactics other than *apply* and *unfold*. For that, we must specify a cost, an optional *pattern* and a *tactic* to execute.

Example

```
Hint Extern 4 (~(_ = _)) => discriminate : core.
```

Now, when the head of the goal is a disequality, `auto` will try discriminate if it does not manage to solve the goal with hints with a cost less than 4.

One can even use some sub-patterns of the pattern in the tactic script. A sub-pattern is a question mark followed by an identifier, like `?X1` or `?X2`. Here is an example:

Example

```
Require Import List.
Hint Extern 5 ({?X1 = ?X2} + {?X1 <> ?X2}) => generalize X1, X2; decide equality : eqdec.
↪
Goal forall a b:list (nat * nat), {a = b} + {a <> b}.
    1 subgoal

      ============================
      forall a b : list (nat * nat), {a = b} + {a <> b}

Info 1 auto with eqdec.
    <ltac_plugin::auto@0> eqdec
    No more subgoals.
```

Variant: `Hint Cut` *regexp* : *ident*

> **Warning:** These hints currently only apply to typeclass proof search and the *typeclasses eauto* tactic.

This command can be used to cut the proof-search tree according to a regular expression matching paths to be cut. The grammar for regular expressions is the following. Beware, there is no operator precedence during parsing, one can check with *Print HintDb* to verify the current cut expression:

```
regexp  ::=  ident (hint or instance identifier)
             _ (any hint)
             regexp | regexp (disjunction)
             regexp regexp (sequence)
             regexp * (Kleene star)
             emp (empty)
             eps (epsilon)
             ( regexp )
```

The `emp` regexp does not match any search path while `eps` matches the empty path. During proof search, the path of successive successful hints on a search branch is recorded, as a list of identifiers for the hints (note that *Hint Extern*'s do not have an associated identifier). Before applying any hint *ident* the current path `p` extended with *ident* is matched against the current cut expression `c` associated to the hint database. If matching succeeds, the hint is *not* applied. The semantics of `Hint Cut` *regexp* is to set the cut expression to `c | regexp`, the initial cut expression being `emp`.

Variant: `Hint Mode` *qualid* `+` `!` `-` `*` : *ident*

This sets an optional mode of use of the identifier *qualid*. When proof-search faces a goal that ends in an application of *qualid* to arguments *term* ... *term*, the mode tells if the hints associated to *qualid* can be applied or not. A mode specification is a list of n `+`, `!` or `-` items that specify if an argument of the identifier is to be treated as an input (`+`), if its head only is an input (`!`) or an output (`-`) of the identifier. For a mode to match a list of arguments, input terms and input heads *must not* contain existential variables or be existential variables respectively, while outputs can be any term. Multiple modes can be declared for a single identifier, in that case only one mode needs to match the arguments for the hints to be applied. The head of a term is understood here as the applicative head, or the match or projection scrutinee's head, recursively, casts being ignored. *Hint Mode* is especially useful for typeclasses, when one does not want to support default instances and avoid ambiguity in general. Setting a parameter of a class as an input forces proof-search to be driven by that index of the class, with `!` giving more flexibility by allowing existentials to still appear deeper in the index but not at its head.

Note:

- One can use a *Hint Extern* with no pattern to do pattern matching on hypotheses using `match goal with` inside the tactic.
- If you want to add hints such as *Hint Transparent*, *Hint Cut*, or *Hint Mode*, for typeclass resolution, do not forget to put them in the `typeclass_instances` hint database.

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 hints declared to belong to this database in each of the various modules currently loaded. Especially, requiring new modules may extend the database. At Coq startup, only the core database is nonempty and can be used.

core This special database is automatically used by `auto`, except when pseudo-database `nocore` is given to `auto`. The core database contains only basic lemmas about negation, conjunction, and so on. Most of the hints in this database come from the Init and Logic directories.

arith This database contains all lemmas about Peano's arithmetic proved in the directories Init and Arith.

zarith contains lemmas about binary signed integers from the directories theories/ZArith. When required, the module Omega also extends the database zarith with a high-cost hint that calls `omega` on equations and inequalities in `nat` or `Z`.

bool contains lemmas about booleans, mostly from directory theories/Bool.

datatypes is for lemmas about lists, streams and so on that are mainly proved in the Lists subdirectory.

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

typeclass_instances contains all the typeclass instances declared in the environment, including those used for `setoid_rewrite`, from the Classes directory.

fset internal database for the implementation of the `FSets` library.

ordered_type lemmas about ordered types (as defined in the legacy `OrderedType` module), mainly used in the `FSets` and `FMaps` libraries.

You are advised not to put your own hints in the core database, but use one or several databases specific to your development.

Command: `Remove Hints` *term*+ `:` *ident*+

This command removes the hints associated to terms *term*+ in databases *ident*+.

Command: `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:

Command: `Print Hint` *ident*

This command displays only tactics associated with *ident* in the hints list. This is independent of the goal being edited, so this command will not fail if no goal is being edited.

Command: `Print Hint *`

This command displays all declared hints.

Command: `Print HintDb` *ident*

This command displays all hints from database *ident*.

Command: `Hint Rewrite` *term*+ `:` *ident*+

This vernacular command adds the terms *term*+ (their types must be equalities) in the rewriting bases *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 *Section mechanism*): 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:

Command: `Hint Rewrite ->` *term*+ `:` *ident*+

This is strictly equivalent to the command above (we only make explicit the orientation which otherwise defaults to ->).

Command: `Hint Rewrite <-` *term*+ `:` *ident*+

Adds the rewriting rules *term*+ with a right-to-left orientation in the bases *ident*+.

Command: `Hint Rewrite` *term*+ `using` *tactic* `:` *ident*+

When the rewriting rules *term*+ in *ident*+ will be used, the tactic `tactic` will be applied to the generated subgoals, the main subgoal excluded.

Command: `Print Rewrite HintDb` *ident*

This command displays all rewrite hints contained in *ident*.

Hint locality

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.`).

As of today, hints only have a binary behavior regarding locality, as described above: either they disappear at the end of a section scope, or they remain global forever. This causes a scalability issue, because hints coming from an unrelated part of the code may badly influence another development. It can be mitigated to some extent thanks to the *Remove Hints* command, but this is a mere workaround and has some limitations (for instance, external hints cannot be removed).

A proper way to fix this issue is to bind the hints to their module scope, as for most of the other objects Coq uses. Hints should only be made available when the module they are defined in is imported, not just required. It is very difficult to change the historical behavior, as it would break a lot of scripts. We propose a smooth transitional path by providing the *Loose Hint Behavior* option which accepts three flags allowing for a fine-grained handling of non-imported hints.

Option: `Loose Hint Behavior` `"Lax"` | `"Warn"` | `"Strict"`

This option accepts three values, which control the behavior of hints w.r.t. *Import*:

- ”Lax”: this is the default, and corresponds to the historical behavior, that is, hints defined outside of a section have a global scope.
- ”Warn”: outputs a warning when a non-imported hint is used. Note that this is an over-approximation, because a hint may be triggered by a run that will eventually fail and backtrack, resulting in the hint not being actually useful for the proof.
- ”Strict”: changes the behavior of an unloaded hint to a immediate fail tactic, allowing to emulate an import-scoped hint mechanism.

Setting implicit automation tactics

Command: `Proof with` *`tactic`*

This command may be used to start a proof. It defines a default tactic to be used each time a tactic command `tactic`$_1$ is ended by `...`. In this case the tactic command typed by the user is equivalent to `tactic`$_1$ `;tactic`.

See also:

`Proof` in *Switching on/off the proof editing mode*.

Variant: `Proof with` *`tactic`* `using` *`ident`*+

Combines in a single line `Proof with` and `Proof using`, see *Switching on/off the proof editing mode*

Variant: `Proof using` *`ident`*+ `with` *`tactic`*

Combines in a single line `Proof with` and `Proof using`, see *Switching on/off the proof editing mode*

5.3.10 Decision procedures

`tauto`

This tactic implements a decision procedure for intuitionistic propositional calculus based on the contraction-free sequent calculi LJT* of Roy Dyckhoff *[Dyc92]*. 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.

Example

The following goal can be proved by *`tauto`* whereas *`auto`* would fail:

```
Goal forall (x:nat) (P:nat -> Prop), x = 0 \/ P x -> x <> 0 -> P x.
    1 subgoal

      ============================
      forall (x : nat) (P : nat -> Prop), x = 0 \/ P x -> x <> 0 -> P x

intros.
    1 subgoal

      x : nat
      P : nat -> Prop
      H : x = 0 \/ P x
      H0 : x <> 0
      ============================
```

(continues on next page)

(continued from previous page)

```
      P x

tauto.
    No more subgoals.
```

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 for:

Example

```
Goal forall (A:Prop) (P:nat -> Prop), A \/ (forall x:nat, ~ A -> P x) -> forall x:nat, ~ A -> P x.
    1 subgoal

      ============================
      forall (A : Prop) (P : nat -> Prop),
      A \/ (forall x : nat, ~ A -> P x) -> forall x : nat, ~ A -> P x

tauto.
    No more subgoals.
```

Note: In contrast, *tauto* cannot solve the following goal `Goal forall (A:Prop) (P:nat -> Prop), A \/ (forall x:nat, ~ A -> P x) -> forall x:nat, ~ ~ (A \/ P x).` because `(forall x:nat, ~ A -> P x)` cannot be treated as atomic and an instantiation of `x` is necessary.

Variant: `dtauto`

While *tauto* recognizes inductively defined connectives isomorphic to the standard connectives `and`, `prod`, `or`, `sum`, `False`, `Empty_set`, `unit`, `True`, *dtauto* also recognizes all inductive types with one constructor and no indices, i.e. record-style connectives.

`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 *[Mun94]*. If this tactic fails on some goals then *intuition* fails. In fact, *tauto* is simply `intuition fail`.

Example

For instance, the tactic `intuition auto` applied to the goal:

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

internally replaces it by the equivalent one:

```
(forall (x:nat), P x), B |- P O
```

and then uses *auto* which completes the proof.

Originally due to César Muñoz, these tactics (*tauto* and *intuition*) have been completely re-engineered by David Delahaye using mainly the tactic language (see *Ltac*). The code is now much shorter and a significant increase in performance has been noticed. The general behavior with respect to dependent types, unfolding and introductions has slightly changed to get clearer semantics. This may lead to some incompatibilities.

Variant: `intuition`

Is equivalent to `intuition auto with *`.

Variant: `dintuition`

While *intuition* recognizes inductively defined connectives isomorphic to the standard connectives `and`, `prod`, `or`, `sum`, `False`, `Empty_set`, `unit`, `True`, *dintuition* also recognizes all inductive types with one constructor and no indices, i.e. record-style connectives.

Flag: `Intuition Negation Unfolding`

Controls whether *intuition* unfolds inner negations which do not need to be unfolded. This flag is on by default.

`rtauto`

The *rtauto* tactic solves propositional tautologies similarly to what *tauto* does. The main difference is that the proof term is built using a reflection scheme applied to a sequent calculus proof of the goal. The search procedure is also implemented using a different technique.

Users should be aware that this difference may result in faster proof-search but slower proof-checking, and *rtauto* might not solve goals that *tauto* would be able to solve (e.g. goals involving universal quantifiers).

Note that this tactic is only available after a `Require Import Rtauto`.

`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.

Option: `Firstorder Solver` *tactic*

The default tactic used by *firstorder* when no rule applies is `auto with *`, it can be reset locally or globally using this option.

Command: `Print Firstorder Solver`

Prints the default tactic used by *firstorder* when no rule applies.

Variant: `firstorder` *tactic*

Tries to solve the goal with *tactic* when no logical rule may apply.

Variant: `firstorder using` *qualid*+

Deprecated since version 8.3: Use the syntax below instead (with commas).

Variant: `firstorder using` *qualid*+,

Adds lemmas *qualid*+, to the proof-search environment. If *qualid* refers to an inductive type, it is the collection of its constructors which are added to the proof-search environment.

Variant: `firstorder with` *ident*+

Adds lemmas from *auto* hint bases *ident*+ to the proof-search environment.

Variant: `firstorder` *tactic* `using` *qualid*+, `with` *ident*+

This combines the effects of the different variants of *firstorder*.

Option: `Firstorder Depth` *num*

This option controls the proof-search depth bound.

`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 includes constructor theory (see *injection* and *discriminate*). If the goal is a non-quantified equality, congruence tries to prove it with non-quantified equalities in the context. Otherwise it tries to infer a discriminable equality from those in the context. Alternatively, congruence tries to prove that a hypothesis is equal to the goal or to the negation of another hypothesis.

congruence is also able to take advantage of hypotheses stating quantified equalities, but you have to provide a bound for the number of extra equalities generated that way. Please note that one of the sides of the equality must contain all the quantified variables in order for congruence to match against it.

Example

```
Theorem T (A:Type) (f:A -> A) (g: A -> A -> A) a b: a=(f a) -> (g b (f a))=(f (f a)) -> (g a b)=(f (g b a)) -> (g a b)=a.
    1 subgoal

      A : Type
      f : A -> A
      g : A -> A -> A
      a, b : A
      ============================
      a = f a -> g b (f a) = f (f a) -> g a b = f (g b a) -> g a b = a

intros.
    1 subgoal

      A : Type
      f : A -> A
      g : A -> A -> A
      a, b : A
      H : a = f a
      H0 : g b (f a) = f (f a)
      H1 : g a b = f (g b a)
      ============================
      g a b = a

congruence.
    No more subgoals.

Qed.
Theorem inj (A:Type) (f:A -> A * A) (a c d: A) : f = pair a -> Some (f c) = Some (f d) -> c=d.
    1 subgoal

      A : Type
      f : A -> A * A
      a, c, d : A
      ============================
      f = pair a -> Some (f c) = Some (f d) -> c = d

intros.
    1 subgoal

      A : Type
      f : A -> A * A
      a, c, d : A
      H : f = pair a
      H0 : Some (f c) = Some (f d)
```

(continues on next page)

(continued from previous page)

```
      ============================
      c = d

congruence.
    No more subgoals.

Qed.
```

Variant: `congruence` *num*

Tries to add at most *num* instances of hypotheses stating quantified equalities to the problem in order to solve it. A bigger value of *num* does not make success slower, only failure. You might consider adding some lemmas as hypotheses using assert in order for *congruence* to use them.

Variant: `congruence with` *term*+

Adds *term*+ to the pool of terms used by *congruence*. This helps in case you have partially applied constructors in your goal.

Error: `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 could not be built in Coq because of dependently-typed functions.

Error: `Goal is solvable by congruence but some arguments are missing. Try congruence with` *term*+`, repla`

The decision procedure could solve the goal with the provision that additional arguments are supplied for some partially applied constructors. Any term of an appropriate type will allow the tactic to successfully solve the goal. Those additional arguments can be given to congruence by filling in the holes in the terms given in the error message, using the *congruence with* variant described above.

Flag: `Congruence Verbose`

This flag makes *congruence* print debug information.

5.3.11 Checking properties of terms

Each of the following tactics acts as the identity if the check succeeds, and results in an error otherwise.

`constr_eq` *term term*

This tactic checks whether its arguments are equal modulo alpha conversion, casts and universe constraints. It may unify universes.

Error: `Not equal.`

Error: `Not equal (due to universes).`

`constr_eq_strict` *term term*

This tactic checks whether its arguments are equal modulo alpha conversion, casts and universe constraints. It does not add new constraints.

Error: `Not equal.`

Error: `Not equal (due to universes).`

`unify` *term term*

This tactic checks whether its arguments are unifiable, potentially instantiating existential variables.

Error: `Unable to unify` *term* `with` *term*`.`

Variant: `unify` *`term`* *`term`* `with` *`ident`*

Unification takes the transparency information defined in the hint database *`ident`* into account (see *the hints databases for auto and eauto*).

`is_evar` *`term`*

This tactic checks whether its argument is a current existential variable. Existential variables are uninstantiated variables generated by *`eapply`* and some other tactics.

Error: `Not an evar.`

`has_evar` *`term`*

This tactic checks whether its argument has an existential variable as a subterm. Unlike context patterns combined with `is_evar`, this tactic scans all subterms, including those under binders.

Error: `No evars.`

`is_var` *`term`*

This tactic checks whether its argument is a variable or hypothesis in the current goal context or in the opened sections.

Error: `Not a variable or hypothesis.`

5.3.12 Equality

`f_equal`

This tactic applies to a goal of the form `f` a$_1$ `...` a$_n$ `= f` a $_1$ `...` a $_n$. Using *`f_equal`* on such a goal leads to subgoals `f=f` and a$_1$ = a $_1$ and so on up to a$_n$ `=` a $_n$. Amongst these subgoals, the simple ones (e.g. provable by *`reflexivity`* or *`congruence`*) are automatically solved by *`f_equal`*.

`reflexivity`

This tactic applies to a goal that 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: `The conclusion is not a substitutive equation.`

Error: `Unable to unify ... with ...`

`symmetry`

This tactic applies to a goal that 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`.

`transitivity` *`term`*

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

Variant: `etransitivity`

This tactic behaves like *`transitivity`*, using a fresh evar instead of a concrete *`term`*.

5.3.13 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 `=`.

`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. It solves goals of the form `{x = y} + {~ x = y}` as well.

`compare` *term* *term*

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

`simplify_eq` *term*

Let *term* be the proof of a statement of conclusion *term* `=` *term*. If *term* and *term* are structurally different (in the sense described for the tactic *discriminate*), then the tactic `simplify_eq` behaves as `discriminate` *term*, otherwise it behaves as `injection` *term*.

Note: If some quantified hypothesis of the goal is named *ident*, then `simplify_eq` *ident* first introduces the hypothesis in the local context using `intros until` *ident*.

Variant: `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.

Variant: `simplify_eq` *term* `with` *bindings_list*

This does the same as `simplify_eq` *term* but using the given bindings to instantiate parameters or hypotheses of *term*.

Variant: `esimplify_eq` *num*

Variant: `esimplify_eq` *term* [`with` *bindings_list*]?

This works the same as *simplify_eq* but if the type of *term*, or the type of the hypothesis referred to by *num*, has uninstantiated parameters, these parameters are left as existential variables.

Variant: `simplify_eq`

If the current goal has form `t1 <> t2`, it behaves as `intro` *ident*`;` `simplify_eq` *ident*.

`dependent rewrite ->` *ident*

This tactic applies to any goal. If *ident* has type `(existT B a b)=(existT B a' b')` in the local context (i.e. each *term* of the equality has a sigma type `{ a:A & (B a)}`) this tactic rewrites `a` into `a'` and `b` into `b'` in the current goal. This tactic works even if `B` is also a sigma type. This kind of equalities between dependent pairs may be derived by the *injection* and *inversion* tactics.

Variant: `dependent rewrite <-` *ident*

Analogous to *dependent rewrite ->* but uses the equality from right to left.

5.3.14 Inversion

`functional inversion` *ident*

functional inversion is a tactic that performs inversion on hypothesis *ident* of the form *qualid* [*term*]+ `=` *term* or *term* `=` *qualid* [*term*]+ where *qualid* must have been defined using Function (see *Advanced recursive functions*). Note that this tactic is only available after a `Require Import FunInd`.

Error: `Hypothesis` *ident* `must contain at least one Function.`

Error: `Cannot find inversion information for hypothesis` *ident*`.`

This error may be raised when some inversion lemma failed to be generated by Function.

Variant: `functional inversion` *num*

This does the same thing as `intros until` *num* followed by `functional inversion` *ident* where *ident* is the identifier for the last introduced hypothesis.

Variant: `functional inversion` *ident* *qualid*

Variant: `functional inversion` *num* *qualid*

If the hypothesis *ident* (or *num*) has a type of the form $\mathit{qualid}_1\ \boxed{\mathit{term}_i}^{+} = \mathit{qualid}_2\ \boxed{\mathit{term}_j}^{+}$ where qualid_1 and qualid_2 are valid candidates to functional inversion, this variant allows choosing which *qualid* is inverted.

5.3.15 Classical tactics

In order to ease the proving process, when the `Classical` module is loaded, a few more tactics are available. Make sure to load the module using the `Require Import` command.

`classical_left`
`classical_right`

These tactics are the analog of *left* and *right* but using classical logic. They can only be used for disjunctions. Use *classical_left* to prove the left part of the disjunction with the assumption that the negation of right part holds. Use *classical_right* to prove the right part of the disjunction with the assumption that the negation of left part holds.

5.3.16 Automating

`btauto`

The tactic *btauto* implements a reflexive solver for boolean tautologies. It solves goals of the form `t = u` where `t` and `u` are constructed over the following grammar:

```
btauto_term  ::=  ident
                  true
                  false
                  orb btauto_term btauto_term
                  andb btauto_term btauto_term
                  xorb btauto_term btauto_term
                  negb btauto_term
                  if btauto_term then btauto_term else btauto_term
```

Whenever the formula supplied is not a tautology, it also provides a counter-example.

Internally, it uses a system very similar to the one of the ring tactic.

Note that this tactic is only available after a `Require Import Btauto`.

Error: `Cannot recognize a boolean equality.`

The goal is not of the form `t = u`. Especially note that *btauto* doesn't introduce variables into the context on its own.

`omega`

The tactic *omega*, due to Pierre Crégut, is an automatic decision procedure for Presburger arithmetic. It solves quantifier-free formulas built with `~`, `\/`, `/\`, `->` on top of equalities, inequalities and disequalities on both the type `nat` of natural numbers and `Z` of binary integers. This tactic must be loaded by the command `Require Import Omega`. See the additional documentation about omega (see Chapter *Omega: a solver for quantifier-free problems in Presburger Arithmetic*).

`ring`

This tactic solves equations upon polynomial expressions of a ring (or semiring) structure. It proceeds by normalizing both hand sides of the equation (w.r.t. associativity, commutativity and distributivity, constant propagation) and comparing syntactically the results.

ring_simplify *term**

This tactic applies the normalization procedure described above to the given terms. The tactic then replaces all occurrences of the terms given in the conclusion of the goal by their normal forms. If no term is given, then the conclusion should be an equation and both hand sides are normalized.

See *The ring and field tactic families* for more information on the tactic and how to declare new ring structures. All declared field structures can be printed with the `Print Rings` command.

field

field_simplify *term**

field_simplify_eq

The field tactic is built on the same ideas as ring: this is a reflexive tactic that solves or simplifies equations in a field structure. The main idea is to reduce a field expression (which is an extension of ring expressions with the inverse and division operations) to a fraction made of two polynomial expressions.

Tactic `field` is used to solve subgoals, whereas `field_simplify` *term*+ replaces the provided terms by their reduced fraction. `field_simplify_eq` applies when the conclusion is an equation: it simplifies both hand sides and multiplies so as to cancel denominators. So it produces an equation without division nor inverse.

All of these 3 tactics may generate a subgoal in order to prove that denominators are different from zero.

See *The ring and field tactic families* for more information on the tactic and how to declare new field structures. All declared field structures can be printed with the Print Fields command.

Example

```
Require Import Reals.
    [Loading ML file newring_plugin.cmxs ... done]
    [Loading ML file r_syntax_plugin.cmxs ... done]
    [Loading ML file zify_plugin.cmxs ... done]
    [Loading ML file micromega_plugin.cmxs ... done]

Goal forall x y:R,
(x * y > 0)%R ->
(x * (1 / x + x / (x + y)))%R =
((- 1 / y) * y * (- x * (x / (x + y)) - 1))%R.
    1 subgoal

      ============================
      forall x y : R,
      (x * y > 0)%R ->
      (x * (1 / x + x / (x + y)))%R = (-1 / y * y * (- x * (x / (x + y)) - 1))%R

intros; field.
    1 subgoal

      x, y : R
      H : (x * y > 0)%R
      ============================
      (x + y)%R <> 0%R /\ y <> 0%R /\ x <> 0%R
```

See also:

File plugins/setoid_ring/RealField.v for an example of instantiation, theory theories/Reals for many examples of use of field.

5.3.17 Non-logical tactics

`cycle` *num*

This tactic puts the *num* first goals at the end of the list of goals. If *num* is negative, it will put the last $|num|$ goals at the beginning of the list.

Example

```
Goal P 1 /\ P 2 /\ P 3 /\ P 4 /\ P 5.
    1 subgoal

      ============================
      P 1 /\ P 2 /\ P 3 /\ P 4 /\ P 5

repeat split.
    5 subgoals

      ============================
      P 1

    subgoal 2 is:
     P 2
    subgoal 3 is:
     P 3
    subgoal 4 is:
     P 4
    subgoal 5 is:
     P 5

all: cycle 2.
    5 subgoals

      ============================
      P 3

    subgoal 2 is:
     P 4
    subgoal 3 is:
     P 5
    subgoal 4 is:
     P 1
    subgoal 5 is:
     P 2

all: cycle -3.
    5 subgoals

      ============================
      P 5

    subgoal 2 is:
     P 1
    subgoal 3 is:
```

(continues on next page)

(continued from previous page)

```
     P 2
    subgoal 4 is:
     P 3
    subgoal 5 is:
     P 4
```

swap *num* *num*

This tactic switches the position of the goals of indices *num* and *num*. If either *num* or *num* is negative then goals are counted from the end of the focused goal list. Goals are indexed from 1, there is no goal with position 0.

Example

```
Goal P 1 /\ P 2 /\ P 3 /\ P 4 /\ P 5.
    1 subgoal

      ============================
      P 1 /\ P 2 /\ P 3 /\ P 4 /\ P 5

repeat split.
    5 subgoals

      ============================
      P 1

    subgoal 2 is:
     P 2
    subgoal 3 is:
     P 3
    subgoal 4 is:
     P 4
    subgoal 5 is:
     P 5

all: swap 1 3.
    5 subgoals

      ============================
      P 3

    subgoal 2 is:
     P 2
    subgoal 3 is:
     P 1
    subgoal 4 is:
     P 4
    subgoal 5 is:
     P 5

all: swap 1 -1.
    5 subgoals

      ============================
      P 5
```

(continues on next page)

(continued from previous page)

```
subgoal 2 is:
 P 2
subgoal 3 is:
 P 1
subgoal 4 is:
 P 4
subgoal 5 is:
 P 3
```

revgoals

This tactics reverses the list of the focused goals.

Example

```
Goal P 1 /\ P 2 /\ P 3 /\ P 4 /\ P 5.
    1 subgoal

      ============================
      P 1 /\ P 2 /\ P 3 /\ P 4 /\ P 5

repeat split.
    5 subgoals

      ============================
      P 1

    subgoal 2 is:
     P 2
    subgoal 3 is:
     P 3
    subgoal 4 is:
     P 4
    subgoal 5 is:
     P 5

all: revgoals.
    5 subgoals

      ============================
      P 5

    subgoal 2 is:
     P 4
    subgoal 3 is:
     P 3
    subgoal 4 is:
     P 2
    subgoal 5 is:
     P 1
```

shelve

This tactic moves all goals under focus to a shelf. While on the shelf, goals will not be focused on. They can be solved by unification, or they can be called back into focus with the command *Unshelve*.

Variant: `shelve_unifiable`

Shelves only the goals under focus that are mentioned in other goals. Goals that appear in the type of other goals can be solved by unification.

Example

```
Goal exists n, n=0.
    1 subgoal

      ============================
      exists n : nat, n = 0

refine (ex_intro _ _ _).
    1 focused subgoal
    (shelved: 1)

      ============================
      ?Goal = 0

all: shelve_unifiable.
reflexivity.
    No more subgoals.
```

Command: `Unshelve`

This command moves all the goals on the shelf (see *shelve*) from the shelf into focus, by appending them to the end of the current list of focused goals.

`unshelve` *tactic*

Performs *tactic*, then unshelves existential variables added to the shelf by the execution of *tactic*, prepending them to the current goal.

`give_up`

This tactic removes the focused goals from the proof. They are not solved, and cannot be solved later in the proof. As the goals are not solved, the proof cannot be closed.

The `give_up` tactic can be used while editing a proof, to choose to write the proof script in a non-sequential order.

5.3.18 Delaying solving unification constraints

`solve_constraints`

Flag: `Solve Unification Constraints`

By default, after each tactic application, postponed typechecking unification problems are resolved using heuristics. Unsetting this flag disables this behavior, allowing tactics to leave unification constraints unsolved. Use the *solve_constraints* tactic at any point to solve the constraints.

5.3.19 Proof maintenance

Experimental. Many tactics, such as *intros*, can automatically generate names, such as ”H0” or ”H1” for a new hypothesis introduced from a goal. Subsequent proof steps may explicitly refer to these names. However, future versions of Coq may not assign names exactly the same way, which could cause the proof to fail because the new names don’t match the explicit references in the proof.

The following ”Mangle Names” settings let users find all the places where proofs rely on automatically generated names, which can then be named explicitly to avoid any incompatibility. These settings cause Coq to generate different names, producing errors for references to automatically generated names.

Flag: `Mangle Names`

When set, generated names use the prefix specified in the following option instead of the default prefix.

Option: `Mangle Names Prefix` *`string`*

Specifies the prefix to use when generating names.

5.3.20 Performance-oriented tactic variants

`change_no_check` *`term`*

For advanced usage. Similar to `change` *`term`*, but as an optimization, it skips checking that *`term`* is convertible to the goal.

Recall that the Coq kernel typechecks proofs again when they are concluded to ensure safety. Hence, using *`change`* checks convertibility twice overall, while *`change_no_check`* can produce ill-typed terms, but checks convertibility only once. Hence, *`change_no_check`* can be useful to speed up certain proof scripts, especially if one knows by construction that the argument is indeed convertible to the goal.

In the following example, *`change_no_check`* replaces `False` by `True`, but `Qed` then rejects the proof, ensuring consistency.

Example

```
Goal False.
    1 subgoal

      ============================
      False

change_no_check True.
    1 subgoal

      ============================
      True

exact I.
    No more subgoals.

Fail Qed.
    The command has indeed failed with message:
    The term "I" has type "True" while it is expected to have type "False".
```

`change_no_check` supports all of `change`’s variants.

Variant: `change_no_check` *`term`* `with` *`term`*’

Variant: `change_no_check` *`term`* `at` *`num`*+ `with` *`term`*’

Variant: `change_no_check` *`term`* [[`at` *`num`*+]? `with` *`term`*]? `in` *`ident`*

Example

```
Goal True -> False.
    1 subgoal

      ============================
      True -> False

intro H.
    1 subgoal

      H : True
      ============================
      False

change_no_check False in H.
    1 subgoal

      H : False
      ============================
      False

exact H.
    No more subgoals.

Fail Qed.
    The command has indeed failed with message:
    The term "fun H : True => H" has type "True -> True"
    while it is expected to have type "True -> False".
```

Variant: `convert_concl_no_check` *`term`*

Deprecated since version 8.11.

Deprecated old name for *`change_no_check`*. Does not support any of its variants.

`exact_no_check` *`term`*

For advanced usage. Similar to `exact` *`term`*, but as an optimization, it skips checking that *`term`* has the goal's type, relying on the kernel check instead. See *`change_no_check`* for more explanations.

Example

```
Goal False.
    1 subgoal

      ============================
      False

exact_no_check I.
    No more subgoals.

Fail Qed.
    The command has indeed failed with message:
    The term "I" has type "True" while it is expected to have type "False".
```

Variant: `vm_cast_no_check` *`term`*

For advanced usage. Similar to `exact_no_check` *term*, but additionally instructs the kernel to use *vm_compute* to compare the goal's type with the *term*'s type.

Example

```
Goal False.
    1 subgoal

      ============================
      False

vm_cast_no_check I.
    No more subgoals.

Fail Qed.
    The command has indeed failed with message:
    The term "I" has type "True" while it is expected to have type "False".
```

Variant: `native_cast_no_check` *term*

for advanced usage. similar to `exact_no_check` *term*, but additionally instructs the kernel to use *native_compute* to compare the goal's type with the *term*'s type.

Example

```
Goal False.
    1 subgoal

      ============================
      False

native_cast_no_check I.
    No more subgoals.

Fail Qed.
    The command has indeed failed with message:
    The term "I" has type "True" while it is expected to have type "False".
```

5.4 Ltac

This chapter documents the tactic language $\mathrm{L}_{\mathrm{tac}}$.

We start by giving the syntax, and next, we present the informal semantics. To learn more about the language and especially about its foundations, please refer to *[Del00]*.

Example: Basic tactic macros

Here are some examples of simple tactic macros that the language lets you write.

```
Ltac reduce_and_try_to_solve := simpl; intros; auto.

Ltac destruct_bool_and_rewrite b H1 H2 :=
  destruct b; [ rewrite H1; eauto | rewrite H2; eauto ].
```

See Section *Examples of using Ltac* for more advanced examples.

5.4.1 Syntax

The syntax of the tactic language is given below. See Chapter *The Gallina specification language* for a description of the BNF metasyntax used in these grammar rules. Various already defined entries will be used in this chapter: entries *num*, *int*, *ident qualid*, *term*, *cpattern* and `tactic` represent respectively natural and integer numbers, identifiers, qualified names, Coq terms, patterns and the atomic tactics described in Chapter *Tactics*.

The syntax of `cpattern` is the same as that of terms, but it is extended with pattern matching metavariables. In *cpattern*, a pattern matching metavariable is represented with the syntax `?`*ident*. The notation `_` can also be used to denote metavariable whose instance is irrelevant. In the notation `?`*ident*, the identifier allows us to keep instantiations and to make constraints whereas `_` shows that we are not interested in what will be matched. On the right hand side of pattern matching clauses, the named metavariables are used without the question mark prefix. There is also a special notation for second-order pattern matching problems: in an applicative pattern of the form `%@?`*ident* *ident*$_1$ … *ident*$_n$, the variable *ident* matches any complex expression with (possible) dependencies in the variables *ident*$_i$ and returns a functional term of the form `fun` *ident*$_1$ … `ident`$_n$ `=>` *term*.

The main entry of the grammar is *ltac_expr*. This language is used in proof mode but it can also be used in toplevel definitions as shown below.

Note:

- The infix tacticals `… || …` , `… + …` , and `… ; …` are associative.

  Example

  If you want that *tactic*$_2$; *tactic*$_3$ be fully run on the first subgoal generated by *tactic*$_1$, before running on the other subgoals, then you should not write *tactic*$_1$`; (`*tactic*$_2$`;` *tactic*$_3$`)` but rather *tactic*$_1$`; [>` *tactic*$_2$`;` *tactic*$_3$ `.. ]`.

- In *tacarg*, there is an overlap between *qualid* as a direct tactic argument and *qualid* as a particular case of *term*. The resolution is done by first looking for a reference of the tactic language and if it fails, for a reference to a term. To force the resolution as a reference of the tactic language, use the form `ltac:(`*qualid*`)`. To force the resolution as a reference to a term, use the syntax `(`*qualid*`)`.

- As shown by the figure, tactical `… || …` binds more than the prefix tacticals *try*, *repeat*, *do* and *abstract* which themselves bind more than the postfix tactical `… ;[ … ]` which binds at the same level as `… ; …` .

  Example

  `try repeat` *tactic*$_1$ `||` *tactic*$_2$`;` *tactic*$_3$`; [` *tactic*+| `];` *tactic*$_4$

  is understood as:

  `((try (repeat (`*tactic*$_1$ `||` *tactic*$_2$`));` *tactic*$_3$`); [` *tactic*+| `]);` *tactic*$_4$

```
ltac_expr       ::=  ltac_expr ; ltac_expr
                     [> ltac_expr | ... | ltac_expr ]
                     ltac_expr ; [ ltac_expr | ... | ltac_expr ]
                     ltac_expr3
ltac_expr3      ::=  do (natural | ident) ltac_expr3
                     progress ltac_expr3
                     repeat ltac_expr3
                     try ltac_expr3
                     once ltac_expr3
                     exactly_once ltac_expr3
                     timeout (natural | ident) ltac_expr3
                     time [string] ltac_expr3
                     only selector: ltac_expr3
                     ltac_expr2
ltac_expr2      ::=  ltac_expr1 || ltac_expr3
                     ltac_expr1 + ltac_expr3
                     tryif ltac_expr1 then ltac_expr1 else ltac_expr1
                     ltac_expr1
ltac_expr1      ::=  fun name ... name => atom
                     let [rec] let_clause with ... with let_clause in atom
                     match goal with context_rule | ... | context_rule end
                     match reverse goal with context_rule | ... | context_rule end
                     match ltac_expr with match_rule | ... | match_rule end
                     lazymatch goal with context_rule | ... | context_rule end
                     lazymatch reverse goal with context_rule | ... | context_rule end
                     lazymatch ltac_expr with match_rule | ... | match_rule end
                     multimatch goal with context_rule | ... | context_rule end
                     multimatch reverse goal with context_rule | ... | context_rule end
                     multimatch ltac_expr with match_rule | ... | match_rule end
                     abstract atom
                     abstract atom using ident
                     first [ ltac_expr | ... | ltac_expr ]
                     solve [ ltac_expr | ... | ltac_expr ]
                     idtac [ message_token ... message_token]
                     fail [natural] [message_token ... message_token]
                     gfail [natural] [message_token ... message_token]
                     fresh [ component … component ]
                     context ident [term]
                     eval redexpr in term
                     type of term
                     constr : term
                     uconstr : term
                     type_term term
                     numgoals
                     guard test
                     assert_fails ltac_expr3
                     assert_succeeds ltac_expr3
                     tactic
                     qualid tacarg ... tacarg
                     atom
atom            ::=  qualid
                     ()
                     int
```

```
                                 ( ltac_expr )
component           ::=    string | qualid
message_token       ::=    string | ident | int
tacarg              ::=    qualid
                           ()
                           ltac : atom
                           term
let_clause          ::=    ident [name ... name] := ltac_expr
context_rule        ::=    context_hyp, ..., context_hyp |- cpattern => ltac_expr
                           cpattern => ltac_expr
                           |- cpattern => ltac_expr
                           _ => ltac_expr
context_hyp         ::=    name : cpattern
                           name := cpattern [: cpattern]
match_rule          ::=    cpattern => ltac_expr
                           context [ident] [ cpattern ] => ltac_expr
                           _ => ltac_expr
test                ::=    int = int
                           int (< | <= | > | >=) int
selector            ::=    [ident]
                           int
                           (int | int - int), ..., (int | int - int)
toplevel_selector   ::=    selector
                           all
                           par
                           !
```

```
top        ::=    [Local] Ltac ltac_def with ... with ltac_def
ltac_def   ::=    ident [ident ... ident] := ltac_expr
                  qualid [ident ... ident] ::= ltac_expr
```

5.4.2 Semantics

Tactic expressions can only be applied in the context of a proof. The evaluation yields either a term, an integer or a tactic. Intermediate results can be terms or integers but the final result must be a tactic which is then applied to the focused goals.

There is a special case for `match goal` expressions of which the clauses evaluate to tactics. Such expressions can only be used as end result of a tactic expression (never as argument of a non-recursive local definition or of an application).

The rest of this section explains the semantics of every construction of $\mathrm{L}_{\mathrm{tac}}$.

Sequence

A sequence is an expression of the following form:

ltac_expr$_1$; *ltac_expr*$_2$

The expression *ltac_expr*$_1$ is evaluated to v_1, which must be a tactic value. The tactic v_1 is applied to the current goal, possibly producing more goals. Then *ltac_expr*$_2$ is evaluated to produce v_2, which must be a tactic value. The tactic v_2 is applied to all the goals produced by the prior application. Sequence is associative.

Local application of tactics

Different tactics can be applied to the different goals using the following form:

`[> ltac_expr*| ]`

The expressions *ltac_expr*$_i$ are evaluated to `v`$_i$, for i = 1, ..., n and all have to be tactics. The `v`$_i$ is applied to the i-th goal, for i = 1, ..., n. It fails if the number of focused goals is not exactly n.

Note: If no tactic is given for the i-th goal, it behaves as if the tactic idtac were given. For instance, `[> | auto]` is a shortcut for `[> idtac | auto ]`.

Variant: `[> ltac_expr_i*| | ltac_expr .. | ltac_expr_j*| ]`

In this variant, *ltac_expr* is used for each goal coming after those covered by the list of *ltac_expr*$_i$ but before those covered by the list of *ltac_expr*$_j$.

Variant: `[> ltac_expr*| | .. | ltac_expr*| ]`

In this variant, idtac is used for the goals not covered by the two lists of *ltac_expr*.

Variant: `[> ltac_expr .. ]`

In this variant, the tactic *ltac_expr* is applied independently to each of the goals, rather than globally. In particular, if there are no goals, the tactic is not run at all. A tactic which expects multiple goals, such as `swap`, would act as if a single goal is focused.

Variant: `ltac_expr_0 ; [ltac_expr_i*| ]`

This variant of local tactic application is paired with a sequence. In this variant, there must be as many *ltac_expr*$_i$ as goals generated by the application of *ltac_expr*$_0$ to each of the individual goals independently. All the above variants work in this form too. Formally, `ltac_expr ; [ ... ]` is equivalent to `[> ltac_expr ; [> ... ] .. ]`.

Goal selectors

We can restrict the application of a tactic to a subset of the currently focused goals with:

`toplevel_selector : ltac_expr`

We can also use selectors as a tactical, which allows to use them nested in a tactic expression, by using the keyword `only`:

Variant: `only selector : ltac_expr`

When selecting several goals, the tactic *ltac_expr* is applied globally to all selected goals.

Variant: `[ident] : ltac_expr`

In this variant, *ltac_expr* is applied locally to a goal previously named by the user (see *Existential variables*).

Variant: `num : ltac_expr`

In this variant, *ltac_expr* is applied locally to the *num*-th goal.

Variant: `num-num+, : ltac_expr`

In this variant, *ltac_expr* is applied globally to the subset of goals described by the given ranges. You can write a single `n` as a shortcut for `n-n` when specifying multiple ranges.

Variant: `all:` *ltac_expr*

In this variant, *ltac_expr* is applied to all focused goals. `all:` can only be used at the toplevel of a tactic expression.

Variant: `!:` *ltac_expr*

In this variant, if exactly one goal is focused, *ltac_expr* is applied to it. Otherwise the tactic fails. `!:` can only be used at the toplevel of a tactic expression.

Variant: `par:` *ltac_expr*

In this variant, *ltac_expr* is applied to all focused goals in parallel. The number of workers can be controlled via the command line option `-async-proofs-tac-j` taking as argument the desired number of workers. Limitations: `par:` only works on goals containing no existential variables and *ltac_expr* must either solve the goal completely or do nothing (i.e. it cannot make some progress). `par:` can only be used at the toplevel of a tactic expression.

Error: `No such goal.`

For loop

There is a for loop that repeats a tactic *num* times:

`do` *num* *ltac_expr*

ltac_expr is evaluated to `v` which must be a tactic value. This 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` *ltac_expr*

ltac_expr is evaluated to `v`. If `v` denotes a tactic, this tactic is applied to each focused goal independently. If the application succeeds, the tactic is applied recursively to all the generated subgoals until it eventually fails. The recursion stops in a subgoal when the tactic has failed *to make progress*. The tactic `repeat` *ltac_expr* itself never fails.

Error catching

We can catch the tactic errors with:

`try` *ltac_expr*

ltac_expr is evaluated to `v` which must be a tactic value. The tactic value `v` is applied to each focused goal independently. If the application of `v` fails in a goal, 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` *ltac_expr*

ltac_expr is evaluated to `v` which must be a tactic value. The tactic value `v` is applied to each focused subgoal independently. If the application of `v` to one of the focused subgoal produced subgoals equal to the initial goals (up to syntactical equality), then an error of level 0 is raised.

Error: `Failed to progress.`

Backtracking branching

We can branch with the following structure:

ltac_expr$_1$ `+` *ltac_expr*$_2$

ltac_expr$_1$ and *ltac_expr*$_2$ are evaluated respectively to v_1 and v_2 which must be tactic values. The tactic value v_1 is applied to each focused goal independently and if it fails or a later tactic fails, then the proof backtracks to the current goal and v_2 is applied.

Tactics can be seen as having several successes. When a tactic fails it asks for more successes of the prior tactics. *ltac_expr*$_1$ `+` *ltac_expr*$_2$ has all the successes of v_1 followed by all the successes of v_2. Algebraically, `(`*ltac_expr*$_1$ `+` *ltac_expr*$_2$`);` *ltac_expr*$_3$ `=` `(`*ltac_expr*$_1$`;` *ltac_expr*$_3$`)` `+` `(`*ltac_expr*$_2$`;` *ltac_expr*$_3$`)`.

Branching is left-associative.

First tactic to work

Backtracking branching may be too expensive. In this case we may restrict to a local, left biased, branching and consider the first tactic to work (i.e. which does not fail) among a panel of tactics:

`first [`*ltac_expr*$^{*}_{|}$`]`

The *ltac_expr*$_i$ are evaluated to v_i and v_i must be tactic values for i = 1, ..., n. Supposing n > 1, `first [`*ltac_expr*$_1$ `| ... |` *ltac_expr*$_n$`]` applies v_1 in each focused goal independently and stops if it succeeds; otherwise it tries to apply v_2 and so on. It fails when there is no applicable tactic. In other words, `first [`*ltac_expr*$_1$ `| ... |` *ltac_expr*$_n$`]` behaves, in each goal, as the first v_i to have *at least* one success.

Error: `No applicable tactic.`

Variant: `first` *ltac_expr*

This is an L_{tac} alias that gives a primitive access to the first tactical as an L_{tac} definition without going through a parsing rule. It expects to be given a list of tactics through a `Tactic Notation`, allowing to write notations of the following form:

Example

```
Tactic Notation "foo" tactic_list(tacs) := first tacs.
```

Left-biased branching

Yet another way of branching without backtracking is the following structure:

ltac_expr$_1$ `||` *ltac_expr*$_2$

ltac_expr$_1$ and *ltac_expr*$_2$ are evaluated respectively to v_1 and v_2 which must be tactic values. The tactic value v_1 is applied in each subgoal independently and if it fails *to progress* then v_2 is applied. *ltac_expr*$_1$ `||` *ltac_expr*$_2$ is equivalent to `first [ progress` *ltac_expr*$_1$ `|` *ltac_expr*$_2$ `]` (except that if it fails, it fails like v_2). Branching is left-associative.

Generalized biased branching

The tactic

`tryif` *ltac_expr*$_1$ `then` *ltac_expr*$_2$ `else` *ltac_expr*$_3$

is a generalization of the biased-branching tactics above. The expression *ltac_expr*$_1$ is evaluated to v$_1$, which is then applied to each subgoal independently. For each goal where v$_1$ succeeds at least once, *ltac_expr*$_2$ is evaluated to v$_2$ which is then applied collectively to the generated subgoals. The v$_2$ tactic can trigger backtracking points in v$_1$: where v$_1$ succeeds at least once, `tryif` *ltac_expr*$_1$ `then` *ltac_expr*$_2$ `else` *ltac_expr*$_3$ is equivalent to v$_1$`;` v$_2$. In each of the goals where v$_1$ does not succeed at least once, *ltac_expr*$_3$ is evaluated in v$_3$ which is is then applied to the goal.

Soft cut

Another way of restricting backtracking is to restrict a tactic to a single success *a posteriori*:

`once` *ltac_expr*

ltac_expr is evaluated to `v` which must be a tactic value. The tactic value `v` is applied but only its first success is used. If `v` fails, `once` *ltac_expr* fails like `v`. If `v` has at least one success, `once` *ltac_expr* succeeds once, but cannot produce more successes.

Checking the successes

Coq provides an experimental way to check that a tactic has *exactly one* success:

`exactly_once` *ltac_expr*

ltac_expr is evaluated to `v` which must be a tactic value. The tactic value `v` is applied if it has at most one success. If `v` fails, `exactly_once` *ltac_expr* fails like `v`. If `v` has a exactly one success, `exactly_once` *ltac_expr* succeeds like `v`. If `v` has two or more successes, `exactly_once` *ltac_expr* fails.

Warning: The experimental status of this tactic pertains to the fact if `v` performs side effects, they may occur in an unpredictable way. Indeed, normally `v` would only be executed up to the first success until backtracking is needed, however exactly_once needs to look ahead to see whether a second success exists, and may run further effects immediately.

Error: **`This tactic has more than one success.`**

Checking the failure

Coq provides a derived tactic to check that a tactic *fails*:

`assert_fails` *ltac_expr*

This behaves like *idtac* if *ltac_expr* fails, and behaves like `fail 0` *ltac_expr* `"succeeds"` if *ltac_expr* has at least one success.

Checking the success

Coq provides a derived tactic to check that a tactic has *at least one* success:

`assert_succeeds` *ltac_expr*

This behaves like `tryif (assert_fails` *ltac_expr*`) then fail 0` *ltac_expr* `"fails" else idtac`.

Solving

We may consider the first to solve (i.e. which generates no subgoal) among a panel of tactics:

`solve [`*ltac_expr* *|`]`

The *ltac_expr*$_i$ are evaluated to v_i and v_i must be tactic values, for i = 1, ..., n. Supposing n > 1, `solve [`*ltac_expr*$_1$ `| ... |` *ltac_expr*$_n$`]` applies v_1 to each goal independently and stops if it succeeds; otherwise it tries to apply v_2 and so on. It fails if there is no solving tactic.

Error: `Cannot solve the goal.`

Variant: `solve` *ltac_expr*

This is an L$_{tac}$ alias that gives a primitive access to the `solve:` tactical. See the `first` tactical for more information.

Identity

The constant `idtac` is the identity tactic: it leaves any goal unchanged but it appears in the proof script.

`idtac` *message_token* *

This prints the given tokens. Strings and integers are printed literally. If a (term) variable is given, its contents are printed.

Failing

`fail`

This 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`*, *`repeat`*, *`match goal`*, or the branching tacticals.

Variant: `fail` *num*

The number is the failure level. If no level is specified, it defaults to 0. The level is used by *`try`*, *`repeat`*, *`match goal`* and the branching tacticals. If 0, it makes *`match goal`* consider the next clause (backtracking). If nonzero, the current *`match goal`* block, *`try`*, *`repeat`*, or branching command is aborted and the level is decremented. In the case of `+`, a nonzero level skips the first backtrack point, even if the call to `fail` *num* is not enclosed in a `+` command, respecting the algebraic identity.

Variant: `fail` *message_token* *

The given tokens are used for printing the failure message.

Variant: `fail` *num* *message_token* *

This is a combination of the previous variants.

Variant: `gfail`

This variant fails even when used after `;` and there are no goals left. Similarly, `gfail` fails even when used after `all:` and there are no goals left. See the example for clarification.

Variant: `gfail` *message_token* *

Variant: `gfail` *num* [*message_token*]*

These variants fail with an error message or an error level even if there are no goals left. Be careful however if Coq terms have to be printed as part of the failure: term construction always forces the tactic into the goals, meaning that if there are no goals when it is evaluated, a tactic call like `let x := H in fail 0 x` will succeed.

Error: `Tactic Failure message (level` *num*`).`

Error: `No such goal.`

Example

```
Goal True.
    1 subgoal

      ============================
      True

Proof.
fail.
    Toplevel input, characters 0-5:
    > fail.
    > ^^^^^
    Error: Tactic failure.

Abort.
Goal True.
    1 subgoal

      ============================
      True

Proof.
trivial; fail.
    No more subgoals.

Qed.
Goal True.
    1 subgoal

      ============================
      True

Proof.
trivial.
    No more subgoals.

fail.
    Toplevel input, characters 0-5:
    > fail.
    > ^^^^^
    Error: No such goal.

Abort.
Goal True.
    1 subgoal
```

(continues on next page)

(continued from previous page)

```
      ============================
      True

Proof.
trivial.
    No more subgoals.

all: fail.
Qed.
Goal True.
    1 subgoal

      ============================
      True

Proof.
gfail.
    Toplevel input, characters 0-6:
    > gfail.
    > ^^^^^^
    Error: Tactic failure.

Abort.
Goal True.
    1 subgoal

      ============================
      True

Proof.
trivial; gfail.
    Toplevel input, characters 0-7:
    > trivial; gfail.
    > ^^^^^^^
    Error: Tactic failure.

Abort.
Goal True.
    1 subgoal

      ============================
      True

Proof.
trivial.
    No more subgoals.

gfail.
    Toplevel input, characters 0-6:
    > gfail.
    > ^^^^^^
    Error: No such goal.

Abort.
Goal True.
    1 subgoal
```

(continues on next page)

(continued from previous page)

```
      ============================
      True

Proof.
trivial.
    No more subgoals.

all: gfail.
    Toplevel input, characters 0-11:
    > all: gfail.
    > ^^^^^^^^^^^
    Error: Tactic failure.

Abort.
```

Timeout

We can force a tactic to stop if it has not finished after a certain amount of time:

`timeout` *`num`* *`ltac_expr`*

`ltac_expr` is evaluated to `v` which must be a tactic value. The tactic value `v` is applied normally, except that it is interrupted after *`num`* seconds if it is still running. In this case the outcome is a failure.

> **Warning:** For the moment, timeout is based on elapsed time in seconds, which is very machine-dependent: a script that works on a quick machine may fail on a slow one. The converse is even possible if you combine a timeout with some other tacticals. This tactical is hence proposed only for convenience during debugging or other development phases, we strongly advise you to not leave any timeout in final scripts. Note also that this tactical isn't available on the native Windows port of Coq.

Timing a tactic

A tactic execution can be timed:

`time` *`string`* *`ltac_expr`*

evaluates *`ltac_expr`* and displays the running time of the tactic expression, whether it fails or succeeds. In case of several successes, the time for each successive run is displayed. Time is in seconds and is machine-dependent. The *`string`* argument is optional. When provided, it is used to identify this particular occurrence of time.

Timing a tactic that evaluates to a term

Tactic expressions that produce terms can be timed with the experimental tactic

`time_constr` *`ltac_expr`*

which evaluates *`ltac_expr`* `()` and displays the time the tactic expression evaluated, assuming successful evaluation. Time is in seconds and is machine-dependent.

This tactic currently does not support nesting, and will report times based on the innermost execution. This is due to the fact that it is implemented using the following internal tactics:

`restart_timer` *string*

Reset a timer

`finish_timing` [`(`*string*`)`]? *string*

Display an optionally named timer. The parenthesized string argument is also optional, and determines the label associated with the timer for printing.

By copying the definition of *time_constr* from the standard library, users can achieve support for a fixed pattern of nesting by passing different *string* parameters to *restart_timer* and *finish_timing* at each level of nesting.

Example

```
Ltac time_constr1 tac :=
  let eval_early := match goal with _ => restart_timer "(depth 1)" end in
  let ret := tac () in
  let eval_early := match goal with _ => finish_timing ( "Tactic evaluation" ) "(depth 1)"␣
↪end in
  ret.
    time_constr1 is defined

Goal True.
    1 subgoal

      ============================
      True

let v := time_constr
       ltac:(fun _ =>
              let x := time_constr1 ltac:(fun _ => constr:(10 * 10)) in
              let y := time_constr1 ltac:(fun _ => eval compute in x) in
              y) in
  pose v.
    Tactic evaluation (depth 1) ran for 0. secs (0.u,0.s)
    Tactic evaluation (depth 1) ran for 0. secs (0.u,0.s)
    Tactic evaluation ran for 0.001 secs (0.u,0.s)
    1 subgoal

      n := 100 : nat
      ============================
      True
```

Local definitions

Local definitions can be done as follows:

`let` *ident*$_1$ `:=` *ltac_expr*$_1$ [`with` *ident*$_i$ `:=` *ltac_expr*$_i$]* `in` *ltac_expr*

each *ltac_expr*$_i$ is evaluated to v_i, then, *ltac_expr* is evaluated by substituting v_i to each occurrence of *ident*$_i$, for i = 1, ..., n. There are no dependencies between the *ltac_expr*$_i$ and the *ident*$_i$.

Local definitions can be made recursive by using `let rec` instead of `let`. In this latter case, the definitions are evaluated lazily so that the rec keyword can be used also in non-recursive cases so as to avoid the eager evaluation of local definitions.

Application

An application is an expression of the following form:

qualid *tacarg*+

The reference *qualid* must be bound to some defined tactic definition expecting at least as many arguments as the provided `tacarg`. The expressions *ltac_expr*$_i$ are evaluated to v_i, for i = 1, ..., n.

Function construction

A parameterized tactic can be built anonymously (without resorting to local definitions) with:

`fun` *ident*+ `=>` *ltac_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` *ltac_expr* `with` *cpattern*$_i$ `=>` *ltac_expr*$_i$ +| `end`

The expression *ltac_expr* is evaluated and should yield a term which is matched against `cpattern`$_1$. The matching is non-linear: if a metavariable occurs more than once, it should match the same expression every time. It is first-order except on the variables of the form `@?id` that occur in head position of an application. For these variables, the matching is second-order and returns a functional term.

Alternatively, when a metavariable of the form `?id` occurs under binders, say x_1, …, x_n and the expression matches, the metavariable is instantiated by a term which can then be used in any context which also binds the variables x_1, …, x_n with same types. This provides with a primitive form of matching under context which does not require manipulating a functional term.

If the matching with *cpattern*$_1$ succeeds, then *ltac_expr*$_1$ is evaluated into some value by substituting the pattern matching instantiations to the metavariables. If *ltac_expr*$_1$ evaluates to a tactic and the match expression is in position to be applied to a goal (e.g. it is not bound to a variable by a `let in`), then this tactic is applied. If the tactic succeeds, the list of resulting subgoals is the result of the match expression. If *ltac_expr*$_1$ does not evaluate to a tactic or if the match expression is not in position to be applied to a goal, then the result of the evaluation of *ltac_expr*$_1$ is the result of the match expression.

If the matching with *cpattern*$_1$ fails, or if it succeeds but the evaluation of *ltac_expr*$_1$ fails, or if the evaluation of *ltac_expr*$_1$ succeeds but returns a tactic in execution position whose execution fails, then `cpattern`$_2$ is used and so on. The pattern _ matches any term and shadows all remaining patterns if any. If all clauses fail (in particular, there is no pattern _) then a no-matching-clause error is raised.

Failures in subsequent tactics do not cause backtracking to select new branches or inside the right-hand side of the selected branch even if it has backtracking points.

Error: `No matching clauses for match.`

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

Error: `Argument of match does not evaluate to a term.`

This happens when *ltac_expr* does not denote a term.

Variant: `multimatch` *ltac_expr* `with` *cpattern*$_i$ `=>` *ltac_expr*$_i$ +| `end`

Using multimatch instead of match will allow subsequent tactics to backtrack into a right-hand side tactic which has backtracking points left and trigger the selection of a new matching branch when all the backtracking points of the right-hand side have been consumed.

The syntax `match …` is, in fact, a shorthand for `once multimatch …`.

Variant: `lazymatch` *ltac_expr* `with` [*cpattern*$_i$ `=>` *ltac_expr*$_i$]+| `end`

Using lazymatch instead of match will perform the same pattern matching procedure but will commit to the first matching branch rather than trying a new matching if the right-hand side fails. If the right-hand side of the selected branch is a tactic with backtracking points, then subsequent failures cause this tactic to backtrack.

Variant: `context` *ident* `[`*cpattern*`]`

This special form of patterns matches any term with a subterm matching cpattern. If there is a match, the optional *ident* is assigned the ”matched context”, i.e. the initial term where the matched subterm is replaced by a hole. The example below will show how to use such term contexts.

If the evaluation of the right-hand-side of a valid match fails, the next matching subterm is tried. If no further subterm matches, the next clause is tried. Matching subterms are considered top-bottom and from left to right (with respect to the raw printing obtained by setting the *Printing All* flag).

Example

```
Ltac f x :=
  match x with
    context f [S ?X] =>
    idtac X;                    (* To display the evaluation order *)
    assert (p := eq_refl 1 : X=1);     (* To filter the case X=1 *)
    let x:= context f[O] in assert (x=O) (* To observe the context *)
  end.
    f is defined

Goal True.
    1 subgoal

      ============================
      True

f (3+4).
    2
    1
    2 subgoals

      p : 1 = 1
      ============================
      1 + 4 = 0

    subgoal 2 is:
     True
```

Pattern matching on goals

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

`match goal with` *context_hyp* + , `|-` *cpattern* `=>` *ltac_expr* + | `| _ =>` *ltac_expr* `end`

If each hypothesis pattern `hyp`$_{1,i}$, with i = 1, ..., m$_1$ is matched (non-linear first-order unification) by a hypothesis of the goal and if `cpattern_1` is matched by the conclusion of the goal, then *ltac_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 the last *ltac_expr* is evaluated to `v` and `v` is applied. Note also that matching against subterms (using the `context` *ident* `[` *cpattern* `]`) is available and is also subject to yielding several matchings.

Failures in subsequent tactics do not cause backtracking to select new branches or combinations of hypotheses, or inside the right-hand side of the selected branch even if it has backtracking points.

Error: `No matching clauses for match goal.`

No clause succeeds, i.e. all matching patterns, if any, fail at the application of the right-hand-side.

Note: 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 right to left (i.e. hyp$_{i,m_i}$‘ before hyp$_{i,1}$). For each hypothesis pattern, the goal hypotheses are matched in order (newest first), but it possible to reverse this order (oldest first) with the `match reverse goal with` variant.

Variant: `multimatch goal with` *context_hyp* + , `|-` *cpattern* `=>` *ltac_expr* + | `| _ =>` *ltac_expr* `end`

Using `multimatch` instead of `match` will allow subsequent tactics to backtrack into a right-hand side tactic which has backtracking points left and trigger the selection of a new matching branch or combination of hypotheses when all the backtracking points of the right-hand side have been consumed.

The syntax `match [reverse] goal …` is, in fact, a shorthand for `once multimatch [reverse] goal …`.

Variant: `lazymatch goal with` *context_hyp* + , `|-` *cpattern* `=>` *ltac_expr* + | `| _ =>` *ltac_expr* `end`

Using lazymatch instead of match will perform the same pattern matching procedure but will commit to the first matching branch with the first matching combination of hypotheses rather than trying a new matching if the right-hand side fails. If the right-hand side of the selected branch is a tactic with backtracking points, then subsequent failures cause this tactic to backtrack.

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* `[`*ltac_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 *ltac_expr*.

Error: `Not a context variable.`

Error: `Unbound context identifier` *`ident`*`.`

Generating fresh hypothesis names

Tactics sometimes have to generate new names for hypothesis. Letting the system decide a name with the intro tactic is not so good since it is very awkward to retrieve the name the system gave. The following expression returns an identifier:

`fresh` *`component`*`*`

It evaluates to an identifier unbound in the goal. This fresh identifier is obtained by concatenating the value of the *`component`*s (each of them is, either a *`qualid`* which has to refer to a (unqualified) name, or directly a name denoted by a *`string`*).

If the resulting name is already used, it is padded with a number so that it becomes fresh. If no component is given, the name is a fresh derivative of the name `H`.

Computing in a constr

Evaluation of a term can be performed with:

`eval` *`redexpr`* `in` *`term`*

where *`redexpr`* is a reduction tactic among *`red`*, *`hnf`*, *`compute`*, *`simpl`*, *`cbv`*, *`lazy`*, *`unfold`*, *`fold`*, *`pattern`*.

Recovering the type of a term

`type of` *`term`*

This tactic returns the type of *`term`*.

Manipulating untyped terms

`uconstr :` *`term`*

The terms built in $\mathrm{L}_{\mathrm{tac}}$ are well-typed by default. It may not be appropriate for building large terms using a recursive $\mathrm{L}_{\mathrm{tac}}$ function: the term has to be entirely type checked at each step, resulting in potentially very slow behavior. It is possible to build untyped terms using $\mathrm{L}_{\mathrm{tac}}$ with the `uconstr :` *`term`* syntax.

`type_term` *`term`*

An untyped term, in $\mathrm{L}_{\mathrm{tac}}$, can contain references to hypotheses or to $\mathrm{L}_{\mathrm{tac}}$ variables containing typed or untyped terms. An untyped term can be type checked using the function type_term whose argument is parsed as an untyped term and returns a well-typed term which can be used in tactics.

Untyped terms built using `uconstr :` can also be used as arguments to the *`refine`* tactic. In that case the untyped term is type checked against the conclusion of the goal, and the holes which are not solved by the typing procedure are turned into new subgoals.

Counting the goals

`numgoals`

The number of goals under focus can be recovered using the `numgoals` function. Combined with the guard command below, it can be used to branch over the number of goals produced by previous tactics.

Example

```
Ltac pr_numgoals := let n := numgoals in idtac "There are" n "goals".
Goal True /\ True /\ True.
split;[|split].

all:pr_numgoals.
    There are 3 goals
```

Testing boolean expressions

guard *test*

The *guard* tactic tests a boolean expression, and fails if the expression evaluates to false. If the expression evaluates to true, it succeeds without affecting the proof.

The accepted tests are simple integer comparisons.

Example

```
Goal True /\ True /\ True.
split;[|split].

all:let n:= numgoals in guard n<4.
Fail all:let n:= numgoals in guard n=2.
    The command has indeed failed with message:
    Condition not satisfied: 3=2
```

Error: `Condition not satisfied.`

Proving a subgoal as a separate lemma

abstract *ltac_expr*

From the outside, `abstract` *ltac_expr* is the same as `solve` *ltac_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. Such an auxiliary lemma is inlined in the final proof term.

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.

It may be useful to generate lemmas minimal w.r.t. the assumptions they depend on. This can be obtained thanks to the option below.

> **Warning:** The abstract tactic, while very useful, still has some known limitations, see https://github.com/coq/coq/issues/9146 for more details. Thus we recommend using it caution in some ”non-standard” contexts. In particular, `abstract` won't properly work when used inside quotations `ltac:(...)`, or if used as part of typeclass resolution, it may produce wrong terms when in universe polymorphic mode.

Variant: `abstract` *`ltac_expr`* `using` *`ident`*

Give explicitly the name of the auxiliary lemma.

Warning: Use this feature at your own risk; explicitly named and reused subterms don't play well with asynchronous proofs.

Variant: `transparent_abstract` *`ltac_expr`*

Save the subproof in a transparent lemma rather than an opaque one.

Warning: Use this feature at your own risk; building computationally relevant terms with tactics is fragile.

Variant: `transparent_abstract` *`ltac_expr`* `using` *`ident`*

Give explicitly the name of the auxiliary transparent lemma.

Warning: Use this feature at your own risk; building computationally relevant terms with tactics is fragile, and explicitly named and reused subterms don't play well with asynchronous proofs.

Error: `Proof is not complete.`

5.4.3 Tactic toplevel definitions

Defining L_{tac} functions

Basically, L_{tac} toplevel definitions are made as follows:

Command: `Local`? `Ltac` *`ident`* *`ident`** `:=` *`ltac_expr`*

This defines a new L_{tac} function that can be used in any tactic script or new L_{tac} toplevel definition.

If preceded by the keyword `Local`, the tactic definition will not be exported outside the current module.

Note: The preceding definition can equivalently be written:

`Ltac` *`ident`* `:= fun` *`ident`*+ `=>` *`ltac_expr`*

Variant: `Ltac` *`ident`* *`ident`** `with` *`ident`* *`ident`** * `:=` *`ltac_expr`*

This syntax allows recursive and mutual recursive function definitions.

Variant: `Ltac` *`qualid`* *`ident`** `::=` *`ltac_expr`*

This syntax *redefines* an existing user-defined tactic.

A previous definition of qualid must exist in the environment. The new definition will always be used instead of the old one and it goes across module boundaries.

Printing L_{tac} tactics

Command: `Print Ltac` *`qualid`*

Defined L_{tac} functions can be displayed using this command.

Command: `Print Ltac Signatures`

This command displays a list of all user-defined tactics, with their arguments.

5.4.4 Examples of using L_{tac}

Proof that the natural numbers have at least two elements

Example: Proof that the natural numbers have at least two elements

The first example shows how to use pattern matching over the proof context to prove that natural numbers have at least two elements. This can be done as follows:

```
Lemma card_nat :
  ~ exists x y : nat, forall z:nat, x = z \/ y = z.
    1 subgoal

      ============================
      ~ (exists x y : nat, forall z : nat, x = z \/ y = z)

Proof.
intros (x & y & Hz).
    1 subgoal

      x, y : nat
      Hz : forall z : nat, x = z \/ y = z
      ============================
      False

destruct (Hz 0), (Hz 1), (Hz 2).
    8 subgoals

      x, y : nat
      Hz : forall z : nat, x = z \/ y = z
      H : x = 0
      H0 : x = 1
      H1 : x = 2
      ============================
      False

    subgoal 2 is:
     False
    subgoal 3 is:
     False
    subgoal 4 is:
     False
    subgoal 5 is:
     False
    subgoal 6 is:
     False
    subgoal 7 is:
```

(continues on next page)

(continued from previous page)

```
     False
    subgoal 8 is:
     False
```

At this point, the *congruence* tactic would finish the job:

```
all: congruence.
    No more subgoals.
```

But for the purpose of the example, let's craft our own custom tactic to solve this:

```
all: match goal with
     | _ : ?a = ?b, _ : ?a = ?c |- _ => assert (b = c) by now transitivity a
     end.
    8 subgoals

      x, y : nat
      Hz : forall z : nat, x = z \/ y = z
      H : x = 0
      H0 : x = 1
      H1 : x = 2
      H2 : 1 = 2
      ============================
      False

    subgoal 2 is:
     False
    subgoal 3 is:
     False
    subgoal 4 is:
     False
    subgoal 5 is:
     False
    subgoal 6 is:
     False
    subgoal 7 is:
     False
    subgoal 8 is:
     False

all: discriminate.
    No more subgoals.
```

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).

Proving that a list is a permutation of a second list

Example: Proving that a list is a permutation of a second list

Let's first define the permutation predicate:

```
Section Sort.
Variable A : Set.
Inductive perm : list A -> list A -> Prop :=
  | perm_refl : forall l, perm l l
  | perm_cons : forall a l0 l1, perm l0 l1 -> perm (a :: l0) (a :: l1)
  | perm_append : forall a l, perm (a :: l) (l ++ a :: nil)
  | perm_trans : forall l0 l1 l2, perm l0 l1 -> perm l1 l2 -> perm l0 l2.
End Sort.
```

Next we define an auxiliary tactic `perm_aux` which takes an argument used to control the recursion depth. This tactic works as follows: If the lists are identical (i.e. convertible), it completes the proof. Otherwise, if the lists have identical heads, it looks at their tails. Finally, if the lists have different heads, it rotates the first list by putting its head at the end.

Every time we perform a rotation, we decrement `n`. When `n` drops down to `1`, we stop performing rotations and we fail. The idea is to give the length of the list as the initial value of `n`. This way of counting the number of rotations will avoid going back to a head that had been considered before.

From Section *Syntax* we know that Ltac has a primitive notion of integers, but they are only used as arguments for primitive tactics and we cannot make computations with them. Thus, instead, we use Coq's natural number type `nat`.

```
Ltac perm_aux n :=
  match goal with
  | |- (perm _ ?l ?l) => apply perm_refl
  | |- (perm _ (?a :: ?l1) (?a :: ?l2)) =>
     let newn := eval compute in (length l1) in
         (apply perm_cons; perm_aux newn)
  | |- (perm ?A (?a :: ?l1) ?l2) =>
     match eval compute in n with
     | 1 => fail
     | _ =>
         let l1' := constr:(l1 ++ a :: nil) in
         (apply (perm_trans A (a :: l1) l1' l2);
         [ apply perm_append | compute; perm_aux (pred n) ])
     end
  end.
```

The main tactic is `solve_perm`. It computes the lengths of the two lists and uses them as arguments to call `perm_aux` if the lengths are equal. (If they aren't, the lists cannot be permutations of each other.)

```
Ltac solve_perm :=
  match goal with
  | |- (perm _ ?l1 ?l2) =>
     match eval compute in (length l1 = length l2) with
     | (?n = ?n) => perm_aux n
     end
  end.
```

And now, here is how we can use the tactic `solve_perm`:

```
    1 subgoal

      ============================
      perm nat (1 :: 2 :: 3 :: nil) (3 :: 2 :: 1 :: nil)
```

```
solve_perm.
    No more subgoals.
    1 subgoal

      ============================
      perm nat (0 :: 1 :: 2 :: 3 :: 4 :: 5 :: 6 :: 7 :: 8 :: 9 :: nil)
        (0 :: 2 :: 4 :: 6 :: 8 :: 9 :: 7 :: 5 :: 3 :: 1 :: nil)

solve_perm.
    No more subgoals.
```

Deciding intuitionistic propositional logic

Pattern matching on goals allows powerful backtracking when returning tactic values. An interesting application is the problem of deciding intuitionistic propositional logic. Considering the contraction-free sequent calculi LJT* of Roy Dyckhoff *[Dyc92]*, it is quite natural to code such a tactic using the tactic language as shown below.

```
Ltac basic :=
match goal with
    | |- True => trivial
    | _ : False |- _ => contradiction
    | _ : ?A |- ?A => assumption
end.

Ltac simplify :=
repeat (intros;
    match goal with
        | H : ~ _ |- _ => red in H
        | H : _ /\ _ |- _ =>
            elim H; do 2 intro; clear H
        | H : _ \/ _ |- _ =>
            elim H; intro; clear H
        | H : ?A /\ ?B -> ?C |- _ =>
            cut (A -> B -> C);
                [ intro | intros; apply H; split; assumption ]
        | H: ?A \/ ?B -> ?C |- _ =>
            cut (B -> C);
                [ cut (A -> C);
                    [ intros; clear H
                    | intro; apply H; left; assumption ]
                | intro; apply H; right; assumption ]
        | H0 : ?A -> ?B, H1 : ?A |- _ =>
            cut B; [ intro; clear H0 | apply H0; assumption ]
        | |- _ /\ _ => split
        | |- ~ _ => red
    end).

Ltac my_tauto :=
  simplify; basic ||
  match goal with
      | H : (?A -> ?B) -> ?C |- _ =>
          cut (B -> C);
              [ intro; cut (A -> B);
                  [ intro; cut C;
                      [ intro; clear H | apply H; assumption ]
```

(continues on next page)

(continued from previous page)

```
                  | clear H ]
              | intro; apply H; intro; assumption ]; my_tauto
      | H : ~ ?A -> ?B |- _ =>
          cut (False -> B);
              [ intro; cut (A -> False);
                  [ intro; cut B;
                      [ intro; clear H | apply H; assumption ]
                  | clear H ]
              | intro; apply H; red; intro; assumption ]; my_tauto
      | |- _ \/ _ => (left; my_tauto) || (right; my_tauto)
  end.
```

The tactic basic tries to reason using simple rules involving truth, falsity and available assumptions. The tactic simplify applies all the reversible rules of Dyckhoff's system. Finally, the tactic my_tauto (the main tactic to be called) simplifies with simplify, tries to conclude with basic 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).

Having defined my_tauto, we can prove tautologies like these:

```
Lemma my_tauto_ex1 :
  forall A B : Prop, A /\ B -> A \/ B.
Proof.
my_tauto.
Qed.
```

```
Lemma my_tauto_ex2 :
  forall A B : Prop, (~ ~ B -> B) -> (A -> B) -> ~ ~ A -> B.
Proof.
my_tauto.
Qed.
```

Deciding type isomorphisms

A trickier problem is to decide equalities between types modulo isomorphisms. Here, we choose to use the isomorphisms of the simply typed λ-calculus with Cartesian product and unit type (see, for example, [dC95]). The axioms of this λ-calculus are given below.

```
Open Scope type_scope.
```

```
Section Iso_axioms.
```

```
Variables A B C : Set.
```

```
Axiom Com : A * B = B * A.
Axiom Ass : A * (B * C) = A * B * C.
Axiom Cur : (A * B -> C) = (A -> B -> C).
Axiom Dis : (A -> B * C) = (A -> B) * (A -> C).
Axiom P_unit : A * unit = A.
Axiom AR_unit : (A -> unit) = unit.
Axiom AL_unit : (unit -> A) = A.
```

```
Lemma Cons : B = C -> A * B = A * C.
Proof.
intro Heq; rewrite Heq; reflexivity.
Qed.

End Iso_axioms.

Ltac simplify_type ty :=
match ty with
    | ?A * ?B * ?C =>
        rewrite <- (Ass A B C); try simplify_type_eq
    | ?A * ?B -> ?C =>
        rewrite (Cur A B C); try simplify_type_eq
    | ?A -> ?B * ?C =>
        rewrite (Dis A B C); try simplify_type_eq
    | ?A * unit =>
        rewrite (P_unit A); try simplify_type_eq
    | unit * ?B =>
        rewrite (Com unit B); try simplify_type_eq
    | ?A -> unit =>
        rewrite (AR_unit A); try simplify_type_eq
    | unit -> ?B =>
        rewrite (AL_unit B); try simplify_type_eq
    | ?A * ?B =>
        (simplify_type A; try simplify_type_eq) ||
        (simplify_type B; try simplify_type_eq)
    | ?A -> ?B =>
        (simplify_type A; try simplify_type_eq) ||
        (simplify_type B; try simplify_type_eq)
end
with simplify_type_eq :=
match goal with
    | |- ?A = ?B => try simplify_type A; try simplify_type B
end.

Ltac len trm :=
match trm with
    | _ * ?B => let succ := len B in constr:(S succ)
    | _ => constr:(1)
end.

Ltac assoc := repeat rewrite <- Ass.

Ltac solve_type_eq n :=
match goal with
    | |- ?A = ?A => reflexivity
    | |- ?A * ?B = ?A * ?C =>
        apply Cons; let newn := len B in solve_type_eq newn
    | |- ?A * ?B = ?C =>
        match eval compute in n with
            | 1 => fail
            | _ =>
                pattern (A * B) at 1; rewrite Com; assoc; solve_type_eq (pred n)
        end
end.
```

```
Ltac compare_structure :=
match goal with
    | |- ?A = ?B =>
        let l1 := len A
        with l2 := len B in
            match eval compute in (l1 = l2) with
                | ?n = ?n => solve_type_eq n
            end
end.

Ltac solve_iso := simplify_type_eq; compare_structure.
```

The tactic to judge equalities modulo this axiomatization is shown above. The algorithm is quite simple. First types are simplified using axioms that can be oriented (this is done by `simplify_type` and `simplify_type_eq`). The normal forms are sequences of Cartesian products without a Cartesian product in the left component. These normal forms are then compared modulo permutation of the components by the tactic `compare_structure`. If they have the same length, the tactic `solve_type_eq` attempts to prove that the types are equal. The main tactic that puts all these components together is `solve_iso`.

Here are examples of what can be solved by `solve_iso`.

```
Lemma solve_iso_ex1 :
  forall A B : Set, A * unit * B = B * (unit * A).
Proof.
intros; solve_iso.
Qed.

Lemma solve_iso_ex2 :
  forall A B C : Set,
    (A * unit -> B * (C * unit)) =
    (A * unit -> (C -> unit) * C) * (unit -> A -> B).
Proof.
intros; solve_iso.
Qed.
```

5.4.5 Debugging L$_{\text{tac}}$ tactics

Backtraces

Flag: `Ltac Backtrace`

Setting this flag displays a backtrace on Ltac failures that can be useful to find out what went wrong. It is disabled by default for performance reasons.

Info trace

Command: `Info` *`num ltac_expr`*

This command can be used to print the trace of the path eventually taken by an L$_{\text{tac}}$ script. That is, the list of executed tactics, discarding all the branches which have failed. To that end the *`Info`* command can be used with the following syntax.

The number *`num`* is the unfolding level of tactics in the trace. At level 0, the trace contains a sequence of tactics in the actual script, at level 1, the trace will be the concatenation of the traces of these tactics, etc…

Example

```
Ltac t x := exists x; reflexivity.
Goal exists n, n=0.

Info 0 t 1||t 0.
    exists  with 0;<ltac_plugin::reflexivity@0>
    No more subgoals.

Undo.

Info 1 t 1||t 0.
    <ltac_plugin::exists@1>  with 0;simple refine ?X12;<unknown>
    No more subgoals.
```

The trace produced by *Info* tries its best to be a reparsable $\mathtt{L}_{\mathrm{tac}}$ script, but this goal is not achievable in all generality. So some of the output traces will contain oddities.

As an additional help for debugging, the trace produced by *Info* contains (in comments) the messages produced by the *idtac* tactical at the right position in the script. In particular, the calls to idtac in branches which failed are not printed.

Option: `Info Level` *num*

This option is an alternative to the *Info* command.

This will automatically print the same trace as `Info` *num* at each tactic call. The unfolding level can be overridden by a call to the *Info* command.

Interactive debugger

Flag: `Ltac Debug`

This flag governs the step-by-step debugger that comes with the $\mathtt{L}_{\mathrm{tac}}$ interpreter.

When the debugger is activated, it stops at every step of the evaluation of the current $\mathtt{L}_{\mathrm{tac}}$ expression and prints information on what it is doing. The debugger stops, prompting for a command which can be one of the following:

simple newline:	go to the next step
h:	get help
x:	exit current evaluation
s:	continue current evaluation without stopping
r n:	advance n steps further
r string:	advance up to the next call to “idtac string”

Error: `Debug mode not available in the IDE`

A non-interactive mode for the debugger is available via the flag:

Flag: `Ltac Batch Debug`

This flag has the effect of presenting a newline at every prompt, when the debugger is on. The debug log thus created, which does not require user input to generate when this flag is set, can then be run through external tools such as diff.

Profiling L_{tac} tactics

It is possible to measure the time spent in invocations of primitive tactics as well as tactics defined in L_{tac} and their inner invocations. The primary use is the development of complex tactics, which can sometimes be so slow as to impede interactive usage. The reasons for the performance degradation can be intricate, like a slowly performing L_{tac} match or a sub-tactic whose performance only degrades in certain situations. The profiler generates a call tree and indicates the time spent in a tactic depending on its calling context. Thus it allows to locate the part of a tactic definition that contains the performance issue.

Flag: `Ltac Profiling`

This flag enables and disables the profiler.

Command: `Show Ltac Profile`

Prints the profile

Variant: `Show Ltac Profile` *`string`*

Prints a profile for all tactics that start with *`string`*. Append a period (.) to the string if you only want exactly that name.

Command: `Reset Ltac Profile`

Resets the profile, that is, deletes all accumulated information.

> **Warning:** Backtracking across a *`Reset Ltac Profile`* will not restore the information.

```
Require Import Coq.omega.Omega.
Ltac mytauto := tauto.
Ltac tac := intros; repeat split; omega || mytauto.
Notation max x y := (x + (y - x)) (only parsing).
Goal forall x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z,
    max x (max y z) = max (max x y) z /\ max x (max y z) = max (max x y) z
    /\
    (A /\ B /\ C /\ D /\ E /\ F /\ G /\ H /\ I /\ J /\ K /\ L /\ M /\
     N /\ O /\ P /\ Q /\ R /\ S /\ T /\ U /\ V /\ W /\ X /\ Y /\ Z
     ->
     Z /\ Y /\ X /\ W /\ V /\ U /\ T /\ S /\ R /\ Q /\ P /\ O /\ N /\
     M /\ L /\ K /\ J /\ I /\ H /\ G /\ F /\ E /\ D /\ C /\ B /\ A).
Proof.
```

```
Set Ltac Profiling.
tac.
    No more subgoals.
```

```
Show Ltac Profile.
    total time:      2.258s

     tactic                                   local  total   calls       max

    tac -----------------------------------   0.1% 100.0%       1    2.258s
    <Coq.Init.Tauto.with_uniform_flags> ---   0.0%  73.3%      26    0.109s
    <Coq.Init.Tauto.tauto_gen> ------------   0.0%  73.3%      26    0.109s
    <Coq.Init.Tauto.tauto_intuitionistic> -   0.0%  73.2%      26    0.109s
    t_tauto_intuit ------------------------   0.1%  73.1%      26    0.109s
    <Coq.Init.Tauto.simplif> --------------  51.0%  70.6%      26    0.107s
    omega ---------------------------------  26.4%  26.4%      28    0.284s
    <Coq.Init.Tauto.is_conj> --------------  12.3%  12.3%   28756    0.017s
    elim id -------------------------------   4.4%   4.4%     650    0.000s
```

(continues on next page)

(continued from previous page)

```
    <Coq.Init.Tauto.axioms> ---------------   1.9%   2.4%       0    0.002s

     tactic                                   local  total   calls       max

    tac -----------------------------------   0.1% 100.0%       1    2.258s
     <Coq.Init.Tauto.with_uniform_flags> -   0.0%  73.3%      26    0.109s
     <Coq.Init.Tauto.tauto_gen> ----------   0.0%  73.2%      26    0.109s
     <Coq.Init.Tauto.tauto_intuitionistic>   0.0%  73.2%      26    0.109s
     t_tauto_intuit ----------------------   0.1%  73.1%      26    0.109s
      <Coq.Init.Tauto.simplif> ----------  51.0%  70.6%      26    0.107s
       <Coq.Init.Tauto.is_conj> --------  12.3%  12.3%   28756    0.017s
       elim id -------------------------   4.4%   4.4%     650    0.000s
      <Coq.Init.Tauto.axioms> -----------   1.9%   2.4%       0    0.002s
     omega -------------------------------  26.4%  26.4%      28    0.284s

Show Ltac Profile "omega".
    total time:      2.258s

     tactic                                   local  total   calls       max

    omega ---------------------------------  26.4%  26.4%      28    0.284s

     tactic                                   local  total   calls       max

Abort.
Unset Ltac Profiling.
```

start ltac profiling

This tactic behaves like *idtac* but enables the profiler.

stop ltac profiling

Similarly to *start ltac profiling*, this tactic behaves like *idtac*. Together, they allow you to exclude parts of a proof script from profiling.

reset ltac profile

This tactic behaves like the corresponding vernacular command and allow displaying and resetting the profile from tactic scripts for benchmarking purposes.

show ltac profile

This tactic behaves like the corresponding vernacular command and allow displaying and resetting the profile from tactic scripts for benchmarking purposes.

show ltac profile *string*

This tactic behaves like the corresponding vernacular command and allow displaying and resetting the profile from tactic scripts for benchmarking purposes.

You can also pass the `-profile-ltac` command line option to `coqc`, which turns the *Ltac Profiling* flag on at the beginning of each document, and performs a *Show Ltac Profile* at the end.

> **Warning:** Note that the profiler currently does not handle backtracking into multi-success tactics, and issues a warning to this effect in many cases when such backtracking occurs.

Run-time optimization tactic

`optimize_heap`

This tactic behaves like `idtac`, except that running it compacts the heap in the OCaml run-time system. It is analogous to the Vernacular command *`Optimize Heap`*.

5.5 Ltac2

The Ltac tactic language is probably one of the ingredients of the success of Coq, yet it is at the same time its Achilles' heel. Indeed, Ltac:

- has often unclear semantics
- is very non-uniform due to organic growth
- lacks expressivity (data structures, combinators, types, ...)
- is slow
- is error-prone and fragile
- has an intricate implementation

Following the need of users who are developing huge projects relying critically on Ltac, we believe that we should offer a proper modern language that features at least the following:

- at least informal, predictable semantics
- a type system
- standard programming facilities (e.g., datatypes)

This new language, called Ltac2, is described in this chapter. It is still experimental but we nonetheless encourage users to start testing it, especially wherever an advanced tactic language is needed. The previous implementation of Ltac, described in the previous chapter, will be referred to as Ltac1.

5.5.1 General design

There are various alternatives to Ltac1, such as Mtac or Rtac for instance. While those alternatives can be quite different from Ltac1, we designed Ltac2 to be as close as reasonably possible to Ltac1, while fixing the aforementioned defects.

In particular, Ltac2 is:

- a member of the ML family of languages, i.e.
  - a call-by-value functional language
  - with effects
  - together with the Hindley-Milner type system
- a language featuring meta-programming facilities for the manipulation of Coq-side terms
- a language featuring notation facilities to help write palatable scripts

We describe more in details each point in the remainder of this document.

5.5.2 ML component

Overview

Ltac2 is a member of the ML family of languages, in the sense that it is an effectful call-by-value functional language, with static typing à la Hindley-Milner (see *[DM82]*). It is commonly accepted that ML constitutes a sweet spot in PL design, as it is relatively expressive while not being either too lax (unlike dynamic typing) nor too strict (unlike, say, dependent types).

The main goal of Ltac2 is to serve as a meta-language for Coq. As such, it naturally fits in the ML lineage, just as the historical ML was designed as the tactic language for the LCF prover. It can also be seen as a general-purpose language, by simply forgetting about the Coq-specific features.

Sticking to a standard ML type system can be considered somewhat weak for a meta-language designed to manipulate Coq terms. In particular, there is no way to statically guarantee that a Coq term resulting from an Ltac2 computation will be well-typed. This is actually a design choice, motivated by backward compatibility with Ltac1. Instead, well-typedness is deferred to dynamic checks, allowing many primitive functions to fail whenever they are provided with an ill-typed term.

The language is naturally effectful as it manipulates the global state of the proof engine. This allows to think of proof-modifying primitives as effects in a straightforward way. Semantically, proof manipulation lives in a monad, which allows to ensure that Ltac2 satisfies the same equations as a generic ML with unspecified effects would do, e.g. function reduction is substitution by a value.

To import Ltac2, use the following command:

```
From Ltac2 Require Import Ltac2.
```

Type Syntax

At the level of terms, we simply elaborate on Ltac1 syntax, which is quite close to OCaml. Types follow the simply-typed syntax of OCaml.

The non-terminal `lident` designates identifiers starting with a lowercase.

```
ltac2_type         ::=   ( ltac2_type, ... , ltac2_type ) ltac2_typeconst
                         ( ltac2_type * ... * ltac2_type )
                         ltac2_type -> ltac2_type
                         ltac2_typevar
ltac2_typeconst    ::=   ( modpath . )* lident
ltac2_typevar      ::=   'lident
ltac2_typeparams   ::=   ( ltac2_typevar, ... , ltac2_typevar )
```

The set of base types can be extended thanks to the usual ML type declarations such as algebraic datatypes and records.

Built-in types include:

- `int`, machine integers (size not specified, in practice inherited from OCaml)
- `string`, mutable strings
- `'a array`, mutable arrays
- `exn`, exceptions
- `constr`, kernel-side terms

- `pattern`, term patterns
- `ident`, well-formed identifiers

Type declarations

One can define new types with the following commands.

Command: `Ltac2 Type` *ltac2_typeparams*? *lident*

This command defines an abstract type. It has no use for the end user and is dedicated to types representing data coming from the OCaml world.

Variant: `Ltac2 Type` `rec`? *ltac2_typeparams*? *lident* `:=` *ltac2_typedef*

This command defines a type with a manifest. There are four possible kinds of such definitions: alias, variant, record and open variant types.

```
ltac2_typedef         ::=   ltac2_type
                            [ ltac2_constructordef | ... | ltac2_constructordef ]
                            { ltac2_fielddef ; ... ; ltac2_fielddef }
                            [ .. ]
ltac2_constructordef  ::=   uident [ ( ltac2_type , ... , ltac2_type ) ]
ltac2_fielddef        ::=   [ mutable ] ident : ltac2_type
```

Aliases are just a name for a given type expression and are transparently unfoldable to it. They cannot be recursive. The non-terminal `uident` designates identifiers starting with an uppercase.

Variants are sum types defined by constructors and eliminated by pattern-matching. They can be recursive, but the `rec` flag must be explicitly set. Pattern matching must be exhaustive.

Records are product types with named fields and eliminated by projection. Likewise they can be recursive if the `rec` flag is set.

Variant: `Ltac2 Type` *ltac2_typeparams*? *ltac2_qualid* `::= [` *ltac2_constructordef* `]`

Open variants are a special kind of variant types whose constructors are not statically defined, but can instead be extended dynamically. A typical example is the standard `exn` type. Pattern matching on open variants must always include a catch-all clause. They can be extended with this command.

Term Syntax

The syntax of the functional fragment is very close to the one of Ltac1, except that it adds a true pattern-matching feature, as well as a few standard constructs from ML.

```
ltac2_var          ::=   lident
ltac2_qualid       ::=   ( modpath . )* lident
ltac2_constructor  ::=   uident
ltac2_term         ::=   ltac2_qualid
                         ltac2_constructor
                         ltac2_term ltac2_term ... ltac2_term
                         fun ltac2_var => ltac2_term
                         let ltac2_var := ltac2_term in ltac2_term
                         let rec ltac2_var := ltac2_term in ltac2_term
```

```
                               match ltac2_term with ltac2_branch ... ltac2_branch end
                               int
                               string
                               ltac2_term ; ltac2_term
                               [| ltac2_term ; ... ; ltac2_term |]
                               ( ltac2_term , ... , ltac2_term )
                               { ltac2_field ltac2_field ... ltac2_field }
                               ltac2_term . ( ltac2_qualid )
                               ltac2_term . ( ltac2_qualid ) := ltac2_term
                               [; ltac2_term ; ... ; ltac2_term ]
                               ltac2_term :: ltac2_term
                               ...
ltac2_branch         ::=       ltac2_pattern => ltac2_term
ltac2_pattern        ::=       ltac2_var
                               _
                               ( ltac2_pattern , ... , ltac2_pattern )
                               ltac2_constructor ltac2_pattern ... ltac2_pattern
                               [ ]
                               ltac2_pattern :: ltac2_pattern
ltac2_field          ::=       ltac2_qualid := ltac2_term
```

In practice, there is some additional syntactic sugar that allows e.g. to bind a variable and match on it at the same time, in the usual ML style.

There is dedicated syntax for list and array literals.

Note: For now, deep pattern matching is not implemented.

Ltac Definitions

Command: `Ltac2` `mutable`$^?$ `rec`$^?$ *lident* `:=` *ltac2_term*

This command defines a new global Ltac2 value.

For semantic reasons, the body of the Ltac2 definition must be a syntactical value, that is, a function, a constant or a pure constructor recursively applied to values.

If `rec` is set, the tactic is expanded into a recursive binding.

If `mutable` is set, the definition can be redefined at a later stage (see below).

Command: `Ltac2 Set` *qualid* `:=` *ltac2_term*

This command redefines a previous `mutable` definition. Mutable definitions act like dynamic binding, i.e. at runtime, the last defined value for this entry is chosen. This is useful for global flags and the like.

Reduction

We use the usual ML call-by-value reduction, with an otherwise unspecified evaluation order. This is a design choice making it compatible with OCaml, if ever we implement native compilation. The expected equations are as follows:

(fun x => t) V t{x := V} (βv)

let x := V in t t{x := V} (let)

match C V_0 ... V with ... | C x_0 ... x => t | ... end t {x := V } (ι)

(t any term, V values, C constructor)

Note that call-by-value reduction is already a departure from Ltac1 which uses heuristics to decide when to evaluate an expression. For instance, the following expressions do not evaluate the same way in Ltac1.

```
foo (idtac; let x := 0 in bar)
```

```
foo (let x := 0 in bar)
```

Instead of relying on the `idtac` idiom, we would now require an explicit thunk to not compute the argument, and `foo` would have e.g. type `(unit -> unit) -> unit`.

```
foo (fun () => let x := 0 in bar)
```

Typing

Typing is strict and follows the Hindley-Milner system. Unlike Ltac1, there are no type casts at runtime, and one has to resort to conversion functions. See notations though to make things more palatable.

In this setting, all the usual argument-free tactics have type `unit -> unit`, but one can return a value of type `t` thanks to terms of type `unit -> t`, or take additional arguments.

Effects

Effects in Ltac2 are straightforward, except that instead of using the standard IO monad as the ambient effectful world, Ltac2 is has a tactic monad.

Note that the order of evaluation of application is *not* specified and is implementation-dependent, as in OCaml.

We recall that the `Proofview.tactic` monad is essentially a IO monad together with backtracking state representing the proof state.

Intuitively a thunk of type `unit -> 'a` can do the following:

- It can perform non-backtracking IO like printing and setting mutable variables
- It can fail in a non-recoverable way
- It can use first-class backtracking. One way to think about this is that thunks are isomorphic to this type: `(unit -> 'a) ~ (unit -> exn + ('a * (exn -> 'a)))` i.e. thunks can produce a lazy list of results where each tail is waiting for a continuation exception.
- It can access a backtracking proof state, consisting among other things of the current evar assignation and the list of goals under focus.

We now describe more thoroughly the various effects in Ltac2.

Standard IO

The Ltac2 language features non-backtracking IO, notably mutable data and printing operations.

Mutable fields of records can be modified using the set syntax. Likewise, built-in types like `string` and `array` feature imperative assignment. See modules `String` and `Array` respectively.

A few printing primitives are provided in the `Message` module, allowing to display information to the user.

Fatal errors

The Ltac2 language provides non-backtracking exceptions, also known as *panics*, through the following primitive in module `Control`:

```
val throw : exn -> 'a
```

Unlike backtracking exceptions from the next section, this kind of error is never caught by backtracking primitives, that is, throwing an exception destroys the stack. This is codified by the following equation, where `E` is an evaluation context:

```
E[throw e]   throw e

(e value)
```

There is currently no way to catch such an exception, which is a deliberate design choice. Eventually there might be a way to catch it and destroy all backtrack and return values.

Backtracking

In Ltac2, we have the following backtracking primitives, defined in the `Control` module:

```
Ltac2 Type 'a result := [ Val ('a) | Err (exn) ].

val zero : exn -> 'a
val plus : (unit -> 'a) -> (exn -> 'a) -> 'a
val case : (unit -> 'a) -> ('a * (exn -> 'a)) result
```

If one views thunks as lazy lists, then `zero` is the empty list and `plus` is list concatenation, while `case` is pattern-matching.

The backtracking is first-class, i.e. one can write `plus (fun () => "x") (fun _ => "y") : string` producing a backtracking string.

These operations are expected to satisfy a few equations, most notably that they form a monoid compatible with sequentialization.:

```
plus t zero   t ()
plus (fun () => zero e) f   f e
plus (plus t f) g   plus t (fun e => plus (f e) g)

case (fun () => zero e)   Err e
case (fun () => plus (fun () => t) f)   Val (t,f)

let x := zero e in u   zero e
let x := plus t f in u   plus (fun () => let x := t in u) (fun e => let x := f e in u)

(t, u, f, g, e values)
```

Goals

A goal is given by the data of its conclusion and hypotheses, i.e. it can be represented as `[Γ   A]`.

The tactic monad naturally operates over the whole proofview, which may represent several goals, including none. Thus, there is no such thing as *the current goal*. Goals are naturally ordered, though.

It is natural to do the same in Ltac2, but we must provide a way to get access to a given goal. This is the role of the `enter` primitive, which applies a tactic to each currently focused goal in turn:

```
val enter : (unit -> unit) -> unit
```

It is guaranteed that when evaluating `enter f`, `f` is called with exactly one goal under focus. Note that `f` may be called several times, or never, depending on the number of goals under focus before the call to `enter`.

Accessing the goal data is then implicit in the Ltac2 primitives, and may panic if the invariants are not respected. The two essential functions for observing goals are given below.:

```
val hyp : ident -> constr
val goal : unit -> constr
```

The two above functions panic if there is not exactly one goal under focus. In addition, `hyp` may also fail if there is no hypothesis with the corresponding name.

5.5.3 Meta-programming

Overview

One of the major implementation issues of Ltac1 is the fact that it is never clear whether an object refers to the object world or the meta-world. This is an incredible source of slowness, as the interpretation must be aware of bound variables and must use heuristics to decide whether a variable is a proper one or referring to something in the Ltac context.

Likewise, in Ltac1, constr parsing is implicit, so that `foo 0` is not `foo` applied to the Ltac integer expression `0` (Ltac does have a notion of integers, though it is not first-class), but rather the Coq term `Datatypes.O`.

The implicit parsing is confusing to users and often gives unexpected results. Ltac2 makes these explicit using quoting and unquoting notation, although there are notations to do it in a short and elegant way so as not to be too cumbersome to the user.

Generic Syntax for Quotations

In general, quotations can be introduced in terms using the following syntax, where `quotentry` is some parsing entry.

```
ltac2_term += ident : ( quotentry )
```

Built-in quotations

The current implementation recognizes the following built-in quotations:

- `ident`, which parses identifiers (type `Init.ident`).
- `constr`, which parses Coq terms and produces an-evar free term at runtime (type `Init.constr`).

- `open_constr`, which parses Coq terms and produces a term potentially with holes at runtime (type `Init.constr` as well).
- `pattern`, which parses Coq patterns and produces a pattern used for term matching (type `Init.pattern`).
- `reference`, which parses either a *qualid* or `&`*ident*. Qualified names are globalized at internalization into the corresponding global reference, while `&id` is turned into `Std.VarRef id`. This produces at runtime a `Std.reference`. There shall be no white space between the ampersand symbol (`&`) and the identifier (*ident*).

The following syntactic sugar is provided for two common cases.

- `@id` is the same as `ident:(id)`
- `'t` is the same as `open_constr:(t)`

Strict vs. non-strict mode

Depending on the context, quotation-producing terms (i.e. `constr` or `open_constr`) are not internalized in the same way. There are two possible modes, the *strict* and the *non-strict* mode.

- In strict mode, all simple identifiers appearing in a term quotation are required to be resolvable statically. That is, they must be the short name of a declaration which is defined globally, excluding section variables and hypotheses. If this doesn't hold, internalization will fail. To work around this error, one has to specifically use the `&` notation.
- In non-strict mode, any simple identifier appearing in a term quotation which is not bound in the global context is turned into a dynamic reference to a hypothesis. That is to say, internalization will succeed, but the evaluation of the term at runtime will fail if there is no such variable in the dynamic context.

Strict mode is enforced by default, such as for all Ltac2 definitions. Non-strict mode is only set when evaluating Ltac2 snippets in interactive proof mode. The rationale is that it is cumbersome to explicitly add `&` interactively, while it is expected that global tactics enforce more invariants on their code.

Term Antiquotations

Syntax

One can also insert Ltac2 code into Coq terms, similarly to what is possible in Ltac1.

`term += ltac2:(` *`ltac2_term`* `)`

Antiquoted terms are expected to have type `unit`, as they are only evaluated for their side-effects.

Semantics

A quoted Coq term is interpreted in two phases, internalization and evaluation.

- Internalization is part of the static semantics, that is, it is done at Ltac2 typing time.
- Evaluation is part of the dynamic semantics, that is, it is done when a term gets effectively computed by Ltac2.

Note that typing of Coq terms is a *dynamic* process occurring at Ltac2 evaluation time, and not at Ltac2 typing time.

Static semantics

During internalization, Coq variables are resolved and antiquotations are type-checked as Ltac2 terms, effectively producing a `glob_constr` in Coq implementation terminology. Note that although it went through the type-checking of **Ltac2**, the resulting term has not been fully computed and is potentially ill-typed as a runtime **Coq** term.

Example

The following term is valid (with type `unit -> constr`), but will fail at runtime:

```
Ltac2 myconstr () := constr:(nat -> 0).
```

Term antiquotations are type-checked in the enclosing Ltac2 typing context of the corresponding term expression.

Example

The following will type-check, with type `constr`.

```
let x := '0 in constr:(1 + ltac2:(exact x))
```

Beware that the typing environment of antiquotations is **not** expanded by the Coq binders from the term.

Example

The following Ltac2 expression will **not** type-check:

```
`constr:(fun x : nat => ltac2:(exact x))`
`(* Error: Unbound variable 'x' *)`
```

There is a simple reason for that, which is that the following expression would not make sense in general.

```
constr:(fun x : nat => ltac2:(clear @x; exact x))
```

Indeed, a hypothesis can suddenly disappear from the runtime context if some other tactic pulls the rug from under you.

Rather, the tactic writer has to resort to the **dynamic** goal environment, and must write instead explicitly that she is accessing a hypothesis, typically as follows.

```
constr:(fun x : nat => ltac2:(exact (hyp @x)))
```

This pattern is so common that we provide dedicated Ltac2 and Coq term notations for it.

- `&x` as an Ltac2 expression expands to `hyp @x`.
- `&x` as a Coq constr expression expands to `ltac2:(Control.refine (fun () => hyp @x))`.

In the special case where Ltac2 antiquotations appear inside a Coq term notation, the notation variables are systematically bound in the body of the tactic expression with type `Ltac2.Init.preterm`. Such a type represents untyped syntactic Coq expressions, which can by typed in the current context using the `Ltac2.Constr.pretype` function.

Example

The following notation is essentially the identity.

```
Notation "[ x ]" := ltac2:(let x := Ltac2.Constr.pretype x in exact $x) (only parsing).
```

Dynamic semantics

During evaluation, a quoted term is fully evaluated to a kernel term, and is in particular type-checked in the current environment.

Evaluation of a quoted term goes as follows.

- The quoted term is first evaluated by the pretyper.
- Antiquotations are then evaluated in a context where there is exactly one goal under focus, with the hypotheses coming from the current environment extended with the bound variables of the term, and the resulting term is fed into the quoted term.

Relative orders of evaluation of antiquotations and quoted term are not specified.

For instance, in the following example, `tac` will be evaluated in a context with exactly one goal under focus, whose last hypothesis is `H : nat`. The whole expression will thus evaluate to the term `fun H : nat => H`.

```
let tac () := hyp @H in constr:(fun H : nat => ltac2:(tac ()))
```

Many standard tactics perform type-checking of their argument before going further. It is your duty to ensure that terms are well-typed when calling such tactics. Failure to do so will result in non-recoverable exceptions.

Trivial Term Antiquotations

It is possible to refer to a variable of type `constr` in the Ltac2 environment through a specific syntax consistent with the antiquotations presented in the notation section.

`term += $`*`lident`*

In a Coq term, writing `$x` is semantically equivalent to `ltac2:(Control.refine (fun () => x))`, up to re-typechecking. It allows to insert in a concise way an Ltac2 variable of type `constr` into a Coq term.

Match over terms

Ltac2 features a construction similar to Ltac1 `match` over terms, although in a less hard-wired way.

```
ltac2_term      ::=   match! ltac2_term with constrmatching .. constrmatching end
                      lazy_match! ltac2_term with constrmatching .. constrmatching end
                      multi_match! ltac2_term with constrmatching .. constrmatching end
constrmatching  ::=   | constrpattern => ltac2_term
constrpattern   ::=   term
                      context [ term ]
                      context lident [ term ]
```

This construction is not primitive and is desugared at parsing time into calls to term matching functions from the `Pattern` module. Internally, it is implemented thanks to a specific scope accepting the *`constrmatching`* syntax.

Variables from the *`constrpattern`* are statically bound in the body of the branch, to values of type `constr` for the variables from the *`term`* pattern and to a value of type `Pattern.context` for the variable *`lident`*.

Note that unlike Ltac, only lowercase identifiers are valid as Ltac2 bindings, so that there will be a syntax error if one of the bound variables starts with an uppercase character.

The semantics of this construction is otherwise the same as the corresponding one from Ltac1, except that it requires the goal to be focused.

Match over goals

Similarly, there is a way to match over goals in an elegant way, which is just a notation desugared at parsing time.

```
ltac2_term    ::=  match! [ reverse ] goal with goalmatching ... goalmatching end
                   lazy_match! [ reverse ] goal with goalmatching ... goalmatching end
                   multi_match! [ reverse ] goal with goalmatching ... goalmatching end
goalmatching  ::=  | [ hypmatching ... hypmatching |- constrpattern ] => ltac2_term
hypmatching   ::=  lident : constrpattern
                   _ : constrpattern
```

Variables from *hypmatching* and *constrpattern* are bound in the body of the branch. Their types are:

- `constr` for pattern variables appearing in a *term*
- `Pattern.context` for variables binding a context
- `ident` for variables binding a hypothesis name.

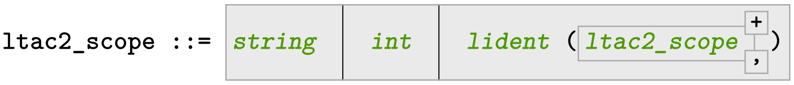
The same identifier caveat as in the case of matching over constr applies, and this features has the same semantics as in Ltac1. In particular, a `reverse` flag can be specified to match hypotheses from the more recently introduced to the least recently introduced one.

5.5.4 Notations

Notations are the crux of the usability of Ltac1. We should be able to recover a feeling similar to the old implementation by using and abusing notations.

Scopes

A scope is a name given to a grammar entry used to produce some Ltac2 expression at parsing time. Scopes are described using a form of S-expression.



A few scopes contain antiquotation features. For the sake of uniformity, all antiquotations are introduced by the syntax `$`*lident*.

The following scopes are built-in.

- `constr`:

  – parses `c = ` *term* and produces `constr:(c)`

  This scope can be parameterized by a list of delimiting keys of interpretation scopes (as described in *Local interpretation rules for notations*), describing how to interpret the parsed term. For instance, `constr(A, B)` parses `c = ` *term* and produces `constr:(c%A%B)`.

- `ident`:
  - parses `id = `*ident* and produces `ident:(id)`
  - parses `$(x = `*ident*`)` and produces the variable `x`
- `list0(`*ltac2_scope*`)`:
  - if *ltac2_scope* parses *quotentry*, then it parses `(`*quotentry*$_0$`, ..., `*quotentry*$_n$`)` and produces `[`*quotentry*$_0$`; ...; `*quotentry*$_n$`]`.
- `list0(`*ltac2_scope*`, sep = `*string*$_{sep}$`)`:
  - if *ltac2_scope* parses *quotentry*, then it parses `(`*quotentry*$_0$ *string*$_{sep}$ `...` *string*$_{sep}$ *quotentry*$_n$`)` and produce `[`*quotentry*$_0$`; ...; `*quotentry*$_n$`]`.
- `list1`: same as `list0` (with or without separator) but parses $\boxed{quotentry}^{+}$ instead of $\boxed{quotentry}^{*}$.
- `opt(`*ltac2_scope*`)`
  - if *ltac2_scope* parses *quotentry*, parses $\boxed{quotentry}^{?}$ and produces either `None` or `Some x` where `x` is the parsed expression.
- `self`:
  - parses a Ltac2 expression at the current level and returns it as is.
- `next`:
  - parses a Ltac2 expression at the next level and returns it as is.
- `tactic(n = `*int*`)`:
  - parses a Ltac2 expression at the provided level `n` and returns it as is.
- `thunk(`*ltac2_scope*`)`:
  - parses the same as `scope`, and if `e` is the parsed expression, returns `fun () => e`.
- `STRING`:
  - parses the corresponding string as an identifier and returns `()`.
- `keyword(s = `*string*`)`:
  - parses the string `s` as a keyword and returns `()`.
- `terminal(s = `*string*`)`:
  - parses the string `s` as a keyword, if it is already a keyword, otherwise as an *ident*. Returns `()`.
- `seq(`*ltac2_scope*$_1$`, ..., `*ltac2_scope*$_2$`)`:
  - parses `scope`$_1$, ..., `scope`$_n$ in this order, and produces a tuple made out of the parsed values in the same order. As an optimization, all subscopes of the form `STRING` are left out of the returned tuple, instead of returning a useless unit value. It is forbidden for the various subscopes to refer to the global entry using `self` or `next`.

A few other specific scopes exist to handle Ltac1-like syntax, but their use is discouraged and they are thus not documented.

For now there is no way to declare new scopes from Ltac2 side, but this is planned.

Notations

The Ltac2 parser can be extended with syntactic notations.

Command: `Ltac2 Notation` [*lident* (*ltac2_scope*) | *string*]+ [: *int*]? := *ltac2_term*

A Ltac2 notation adds a parsing rule to the Ltac2 grammar, which is expanded to the provided body where every token from the notation is let-bound to the corresponding generated expression.

Example

Assume we perform:

```
Ltac2 Notation "foo" c(thunk(constr)) ids(list0(ident)) := Bar.f c ids.
```

Then the following expression

```
let y := @X in foo (nat -> nat) x $y
```

will expand at parsing time to

```
let y := @X in  let c := fun () => constr:(nat -> nat) with ids := [@x; y] in Bar.f c
ids
```

Beware that the order of evaluation of multiple let-bindings is not specified, so that you may have to resort to thunking to ensure that side-effects are performed at the right time.

Abbreviations

Variant: `Ltac2 Notation` *lident* := *ltac2_term*

This command introduces a special kind of notation, called an abbreviation, that is designed so that it does not add any parsing rules. It is similar in spirit to Coq abbreviations, insofar as its main purpose is to give an absolute name to a piece of pure syntax, which can be transparently referred to by this name as if it were a proper definition.

The abbreviation can then be manipulated just as a normal Ltac2 definition, except that it is expanded at internalization time into the given expression. Furthermore, in order to make this kind of construction useful in practice in an effectful language such as Ltac2, any syntactic argument to an abbreviation is thunked on-the-fly during its expansion.

For instance, suppose that we define the following.

```
Ltac2 Notation foo := fun x => x ().
```

Then we have the following expansion at internalization time.

```
foo 0   (fun x => x ()) (fun _ => 0)
```

Note that abbreviations are not typechecked at all, and may result in typing errors after expansion.

5.5.5 Evaluation

Ltac2 features a toplevel loop that can be used to evaluate expressions.

Command: `Ltac2 Eval` *ltac2_term*

This command evaluates the term in the current proof if there is one, or in the global environment otherwise, and displays the resulting value to the user together with its type. This command is pure in the sense that it does not modify the state of the proof, and in particular all side-effects are discarded.

5.5.6 Debug

Flag: `Ltac2 Backtrace`

When this flag is set, toplevel failures will be printed with a backtrace.

5.5.7 Compatibility layer with Ltac1

Ltac1 from Ltac2

Simple API

One can call Ltac1 code from Ltac2 by using the `ltac1` quotation. It parses a Ltac1 expression, and semantics of this quotation is the evaluation of the corresponding code for its side effects. In particular, it cannot return values, and the quotation has type `unit`.

```
ltac2_term  ::=  ltac1 : ( ltac_expr )
```

Ltac1 **cannot** implicitly access variables from the Ltac2 scope, but this can be done with an explicit annotation on the `ltac1` quotation.

```
ltac2_term  ::=  ltac1 : ( ident ... ident |- ltac_expr )
```

The return type of this expression is a function of the same arity as the number of identifiers, with arguments of type `Ltac2.Ltac1.t` (see below). This syntax will bind the variables in the quoted Ltac1 code as if they had been bound from Ltac1 itself. Similarly, the arguments applied to the quotation will be passed at runtime to the Ltac1 code.

Low-level API

There exists a lower-level FFI into Ltac1 that is not recommended for daily use, which is available in the `Ltac2.Ltac1` module. This API allows to directly manipulate dynamically-typed Ltac1 values, either through the function calls, or using the `ltac1val` quotation. The latter parses the same as `ltac1`, but has type `Ltac2.Ltac1.t` instead of `unit`, and dynamically behaves as an Ltac1 thunk, i.e. `ltac1val:(foo)` corresponds to the tactic closure that Ltac1 would generate from `idtac; foo`.

Due to intricate dynamic semantics, understanding when Ltac1 value quotations focus is very hard. This is why some functions return a continuation-passing style value, as it can dispatch dynamically between focused and unfocused behaviour.

The same mechanism for explicit binding of variables as described in the previous section applies.

Ltac2 from Ltac1

Same as above by switching Ltac1 by Ltac2 and using the `ltac2` quotation instead.

```
ltac_expr  ::=  ltac2 : ( ltac2_term )
                ltac2 : ( ident ... ident |- ltac2_term )
```

The typing rules are dual, that is, the optional identifiers are bound with type `Ltac2.Ltac1.t` in the Ltac2 expression, which is expected to have type unit. The value returned by this quotation is an Ltac1 function with the same arity as the number of bound variables.

Note that when no variables are bound, the inner tactic expression is evaluated eagerly, if one wants to use it as an argument to a Ltac1 function, one has to resort to the good old `idtac; ltac2:(foo)` trick. For instance, the code below will fail immediately and won't print anything.

```
From Ltac2 Require Import Ltac2.
Set Default Proof Mode "Classic".


Ltac mytac tac := idtac "I am being evaluated"; tac.
    mytac is defined

Goal True.
    1 subgoal

      ============================
      True

Proof.
(* Doesn't print anything *)
Fail mytac ltac2:(fail).
    The command has indeed failed with message:
    Uncaught Ltac2 exception: Tactic_failure (None)

(* Prints and fails *)
Fail mytac ltac:(idtac; ltac2:(fail)).
    I am being evaluated
    The command has indeed failed with message:
    Uncaught Ltac2 exception: Tactic_failure (None)
```

In any case, the value returned by the fully applied quotation is an unspecified dummy Ltac1 closure and should not be further used.

Switching between Ltac languages

We recommend using the *`Default Proof Mode`* option to switch between tactic languages with a proof-based granularity. This allows to incrementally port the proof scripts.

5.5.8 Transition from Ltac1

Owing to the use of a lot of notations, the transition should not be too difficult. In particular, it should be possible to do it incrementally. That said, we do *not* guarantee you it is going to be a blissful walk either. Hopefully, owing to the fact Ltac2 is typed, the interactive dialogue with Coq will help you.

We list the major changes and the transition strategies hereafter.

Syntax changes

Due to conflicts, a few syntactic rules have changed.

- The dispatch tactical `tac; [foo|bar]` is now written `tac > [foo|bar]`.
- Levels of a few operators have been revised. Some tacticals now parse as if they were normal functions. Parentheses are now required around complex arguments, such as abstractions. The tacticals affected are: `try`, `repeat`, `do`, `once`, `progress`, `time`, `abstract`.

- `idtac` is no more. Either use `()` if you expect nothing to happen, `(fun () => ())` if you want a thunk (see next section), or use printing primitives from the `Message` module if you want to display something.

Tactic delay

Tactics are not magically delayed anymore, neither as functions nor as arguments. It is your responsibility to thunk them beforehand and apply them at the call site.

A typical example of a delayed function:

```
Ltac foo := blah.
```

becomes

```
Ltac2 foo () := blah.
```

All subsequent calls to `foo` must be applied to perform the same effect as before.

Likewise, for arguments:

```
Ltac bar tac := tac; tac; tac.
```

becomes

```
Ltac2 bar tac := tac (); tac (); tac ().
```

We recommend the use of syntactic notations to ease the transition. For instance, the first example can alternatively be written as:

```
Ltac2 foo0 () := blah. Ltac2 Notation foo := foo0 ().
```

This allows to keep the subsequent calls to the tactic as-is, as the expression `foo` will be implicitly expanded everywhere into `foo0 ()`. Such a trick also works for arguments, as arguments of syntactic notations are implicitly thunked. The second example could thus be written as follows.

```
Ltac2 bar0 tac := tac (); tac (); tac (). Ltac2 Notation bar := bar0.
```

Variable binding

Ltac1 relies on complex dynamic trickery to be able to tell apart bound variables from terms, hypotheses, etc. There is no such thing in Ltac2, as variables are recognized statically and other constructions do not live in the same syntactic world. Due to the abuse of quotations, it can sometimes be complicated to know what a mere identifier represents in a tactic expression. We recommend tracking the context and letting the compiler print typing errors to understand what is going on.

We list below the typical changes one has to perform depending on the static errors produced by the typechecker.

In Ltac expressions

Error: `Unbound value | constructor X`

- if `X` is meant to be a term from the current stactic environment, replace the problematic use by `'X`.
- if `X` is meant to be a hypothesis from the goal context, replace the problematic use by `&X`.

In quotations

Error: `The reference X was not found in the current environment`

- if `X` is meant to be a tactic expression bound by a Ltac2 let or function, replace the problematic use by `$X`.
- if `X` is meant to be a hypothesis from the goal context, replace the problematic use by `&X`.

Exception catching

Ltac2 features a proper exception-catching mechanism. For this reason, the Ltac1 mechanism relying on `fail` taking integers, and tacticals decreasing it, has been removed. Now exceptions are preserved by all tacticals, and it is your duty to catch them and re-raise them as needed.

5.6 Detailed examples of tactics

This chapter presents detailed examples of certain tactics, to illustrate their behavior.

5.6.1 dependent induction

The tactics `dependent induction` and `dependent destruction` are another solution for inverting inductive predicate instances and potentially doing induction at the same time. It is based on the `BasicElim` tactic of Conor McBride which works by abstracting each argument of an inductive instance by a variable and constraining it by equalities afterwards. This way, the usual induction and destruct tactics can be applied to the abstracted instance and after simplification of the equalities we get the expected goals.

The abstracting tactic is called generalize_eqs and it takes as argument a hypothesis to generalize. It uses the JMeq datatype defined in Coq.Logic.JMeq, hence we need to require it before. For example, revisiting the first example of the inversion documentation:

```
Require Import Coq.Logic.JMeq.
Inductive Le : nat -> nat -> Set :=
     | LeO : forall n:nat, Le 0 n
     | LeS : forall n m:nat, Le n m -> Le (S n) (S m).
Parameter P : nat -> nat -> Prop.
Goal forall n m:nat, Le (S n) m -> P n m.
intros n m H.
```

```
generalize_eqs H.
    1 subgoal

      n, m, gen_x : nat
      H : Le gen_x m
      ============================
      gen_x = S n -> P n m
```

The index `S n` gets abstracted by a variable here, but a corresponding equality is added under the abstract instance so that no information is actually lost. The goal is now almost amenable to do induction or case analysis. One should indeed first move `n` into the goal to strengthen it before doing induction, or `n` will be fixed in the inductive hypotheses (this does not matter for case analysis). As a rule of thumb, all the variables that appear inside constructors in the indices of the hypothesis should be generalized. This is exactly what the `generalize_eqs_vars` variant does:

```
generalize_eqs_vars H.
induction H.
    2 subgoals

      n, n0 : nat
      ============================
      0 = S n -> P n n0

    subgoal 2 is:
     S n0 = S n -> P n (S m)
```

As the hypothesis itself did not appear in the goal, we did not need to use an heterogeneous equality to relate the new hypothesis to the old one (which just disappeared here). However, the tactic works just as well in this case, e.g.:

```
Parameter Q : forall (n m : nat), Le n m -> Prop.
Goal forall n m (p : Le (S n) m), Q (S n) m p.


intros n m p.
    1 subgoal

      n, m : nat
      p : Le (S n) m
      ============================
      Q (S n) m p

generalize_eqs_vars p.
    1 subgoal

      m, gen_x : nat
      p : Le gen_x m
      ============================
      forall (n : nat) (p0 : Le (S n) m), gen_x = S n -> p ~= p0 -> Q (S n) m p0
```

One drawback of this approach is that in the branches one will have to substitute the equalities back into the instance to get the right assumptions. Sometimes injection of constructors will also be needed to recover the needed equalities. Also, some subgoals should be directly solved because of inconsistent contexts arising from the constraints on indexes. The nice thing is that we can make a tactic based on discriminate, injection and variants of substitution to automatically do such simplifications (which may involve the axiom K). This is what the `simplify_dep_elim` tactic from `Coq.Program.Equality` does. For example, we might simplify the previous goals considerably:

```
induction p ; simplify_dep_elim.
    1 subgoal

      n, m : nat
      p : Le n m
      IHp : forall (n0 : nat) (p0 : Le (S n0) m),
            n = S n0 -> p ~= p0 -> Q (S n0) m p0
      ============================
      Q (S n) (S m) (LeS n m p)
```

The higher-order tactic `do_depind` defined in `Coq.Program.Equality` takes a tactic and combines the building blocks we have seen with it: generalizing by equalities calling the given tactic with the generalized induction hypothesis as argument and cleaning the subgoals with respect to equalities. Its most important instantiations are `dependent induction` and `dependent destruction` that do induction or simply case analysis on the generalized hypothesis. For example we can redo what we've done manually with dependent destruction:

```
Lemma ex : forall n m:nat, Le (S n) m -> P n m.

intros n m H.

dependent destruction H.
    1 subgoal

      n, m : nat
      H : Le n m
      ============================
      P n (S m)
```

This gives essentially the same result as inversion. Now if the destructed hypothesis actually appeared in the goal, the tactic would still be able to invert it, contrary to dependent inversion. Consider the following example on vectors:

```
Set Implicit Arguments.

Parameter A : Set.

Inductive vector : nat -> Type :=
         | vnil : vector 0
         | vcons : A -> forall n, vector n -> vector (S n).

Goal forall n, forall v : vector (S n),
         exists v' : vector n, exists a : A, v = vcons a v'.

intros n v.

dependent destruction v.
    1 subgoal

      n : nat
      a : A
      v : vector n
      ============================
      exists (v' : vector n) (a0 : A), vcons a v = vcons a0 v'
```

In this case, the `v` variable can be replaced in the goal by the generalized hypothesis only when it has a type of the form `vector (S n)`, that is only in the second case of the destruct. The first one is dismissed because `S n <> 0`.

A larger example

Let's see how the technique works with induction on inductive predicates on a real example. We will develop an example application to the theory of simply-typed lambda-calculus formalized in a dependently-typed style:

```
Inductive type : Type :=
         | base : type
         | arrow : type -> type -> type.
```

```
Notation " t --> t' " := (arrow t t') (at level 20, t' at next level).

Inductive ctx : Type :=
         | empty : ctx
         | snoc : ctx -> type -> ctx.

Notation " G , tau " := (snoc G tau) (at level 20, tau at next level).

Fixpoint conc (G D : ctx) : ctx :=
         match D with
         | empty => G
         | snoc D' x => snoc (conc G D') x
         end.

Notation " G ; D " := (conc G D) (at level 20).

Inductive term : ctx -> type -> Type :=
         | ax : forall G tau, term (G, tau) tau
         | weak : forall G tau,
                    term G tau -> forall tau', term (G, tau') tau
         | abs : forall G tau tau',
                   term (G , tau) tau' -> term G (tau --> tau')
         | app : forall G tau tau',
                   term G (tau --> tau') -> term G tau -> term G tau'.
```

We have defined types and contexts which are snoc-lists of types. We also have a `conc` operation that concatenates two contexts. The `term` datatype represents in fact the possible typing derivations of the calculus, which are isomorphic to the well-typed terms, hence the name. A term is either an application of:

- the axiom rule to type a reference to the first variable in a context
- the weakening rule to type an object in a larger context
- the abstraction or lambda rule to type a function
- the application to type an application of a function to an argument

Once we have this datatype we want to do proofs on it, like weakening:

```
Lemma weakening : forall G D tau, term (G ; D) tau ->
                  forall tau', term (G , tau' ; D) tau.
```

The problem here is that we can't just use induction on the typing derivation because it will forget about the `G ; D` constraint appearing in the instance. A solution would be to rewrite the goal as:

```
Lemma weakening' : forall G' tau, term G' tau ->
                   forall G D, (G ; D) = G' ->
                   forall tau', term (G, tau' ; D) tau.
```

With this proper separation of the index from the instance and the right induction loading (putting `G` and `D` after the inducted-on hypothesis), the proof will go through, but it is a very tedious process. One is also forced to make a wrapper lemma to get back the more natural statement. The `dependent induction` tactic alleviates this trouble by doing all of this plumbing of generalizing and substituting back automatically. Indeed we can simply write:

```
Require Import Coq.Program.Tactics.
Require Import Coq.Program.Equality.
```

```
Lemma weakening : forall G D tau, term (G ; D) tau ->
                  forall tau', term (G , tau' ; D) tau.

Proof with simpl in * ; simpl_depind ; auto.

intros G D tau H.
dependent induction H generalizing G D ; intros.
```

This call to dependent induction has an additional arguments which is a list of variables appearing in the instance that should be generalized in the goal, so that they can vary in the induction hypotheses. By default, all variables appearing inside constructors (except in a parameter position) of the instantiated hypothesis will be generalized automatically but one can always give the list explicitly.

```
Show.
    4 subgoals

      G0 : ctx
      tau : type
      G, D : ctx
      x : G0, tau = G; D
      tau' : type
      ============================
      term ((G, tau'); D) tau

    subgoal 2 is:
     term ((G, tau'0); D) tau
    subgoal 3 is:
     term ((G, tau'0); D) (tau --> tau')
    subgoal 4 is:
     term ((G, tau'0); D) tau'
```

The `simpl_depind` tactic includes an automatic tactic that tries to simplify equalities appearing at the beginning of induction hypotheses, generally using trivial applications of `reflexivity`. In cases where the equality is not between constructor forms though, one must help the automation by giving some arguments, using the `specialize` tactic for example.

```
destruct D... apply weak; apply ax.
apply ax.

destruct D...


Show.
    4 subgoals

      G0 : ctx
      tau : type
      H : term G0 tau
      tau' : type
      IHterm : forall G D : ctx,
               G0 = G; D -> forall tau' : type, term ((G, tau'); D) tau
      tau'0 : type
      ============================
      term ((G0, tau'), tau'0) tau

    subgoal 2 is:
     term (((G, tau'0); D), t) tau
```

(continues on next page)

(continued from previous page)

```
    subgoal 3 is:
     term ((G, tau'0); D) (tau --> tau')
    subgoal 4 is:
     term ((G, tau'0); D) tau'
```

```
specialize (IHterm G0 empty eq_refl).
    4 subgoals

      G0 : ctx
      tau : type
      H : term G0 tau
      tau' : type
      IHterm : forall tau' : type, term ((G0, tau'); empty) tau
      tau'0 : type
      ============================
      term ((G0, tau'), tau'0) tau

    subgoal 2 is:
     term (((G, tau'0); D), t) tau
    subgoal 3 is:
     term ((G, tau'0); D) (tau --> tau')
    subgoal 4 is:
     term ((G, tau'0); D) tau'
```

Once the induction hypothesis has been narrowed to the right equality, it can be used directly.

```
apply weak, IHterm.
    3 subgoals

      tau : type
      G, D : ctx
      IHterm : forall G0 D0 : ctx,
               G; D = G0; D0 -> forall tau' : type, term ((G0, tau'); D0) tau
      H : term (G; D) tau
      t, tau'0 : type
      ============================
      term (((G, tau'0); D), t) tau

    subgoal 2 is:
     term ((G, tau'0); D) (tau --> tau')
    subgoal 3 is:
     term ((G, tau'0); D) tau'
```

Now concluding this subgoal is easy.

```
constructor; apply IHterm; reflexivity.
```

See also:

The *induction*, *case*, and *inversion* tactics.

5.6.2 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: Ackermann function

```
Require Import Arith.

Parameter Ack : nat -> nat -> nat.

Axiom Ack0 : forall m:nat, Ack 0 m = S m.
Axiom Ack1 : forall n:nat, Ack (S n) 0 = Ack n 1.
Axiom Ack2 : forall n m:nat, Ack (S n) (S m) = Ack n (Ack (S n) m).

Hint Rewrite Ack0 Ack1 Ack2 : base0.

Lemma ResAck0 : Ack 3 2 = 29.
    1 subgoal

      ============================
      Ack 3 2 = 29

autorewrite with base0 using try reflexivity.
    No more subgoals.
```

Example: MacCarthy function

```
Require Import Omega.

Parameter g : nat -> nat -> nat.

Axiom g0 : forall m:nat, g 0 m = m.
Axiom g1 : forall n m:nat, (n > 0) -> (m > 100) -> g n m = g (pred n) (m - 10).
Axiom g2 : forall n m:nat, (n > 0) -> (m <= 100) -> g n m = g (S n) (m + 11).

Hint Rewrite g0 g1 g2 using omega : base1.

Lemma Resg0 : g 1 110 = 100.

    1 subgoal

      ============================
      g 1 110 = 100

autorewrite with base1 using reflexivity || simpl.
    No more subgoals.

Lemma Resg1 : g 1 95 = 91.
    1 subgoal

      ============================
      g 1 95 = 91

autorewrite with base1 using reflexivity || simpl.
    No more subgoals.
```

5.7 The SSReflect proof language

Authors Georges Gonthier, Assia Mahboubi, Enrico Tassi

5.7.1 Introduction

This chapter describes a set of tactics known as SSReflect originally designed to provide support for the so-called *small scale reflection* proof methodology. Despite the original purpose this set of tactic is of general interest and is available in Coq starting from version 8.7.

SSReflect was developed independently of the tactics described in Chapter *Tactics*. Indeed the scope of the tactics part of SSReflect largely overlaps with the standard set of tactics. Eventually the overlap will be reduced in future releases of Coq.

Proofs written in SSReflect typically look quite different from the ones written using only tactics as per Chapter *Tactics*. We try to summarise here the most “visible” ones in order to help the reader already accustomed to the tactics described in Chapter *Tactics* to read this chapter.

The first difference between the tactics described in this chapter and the tactics described in Chapter *Tactics* is the way hypotheses are managed (we call this *bookkeeping*). In Chapter *Tactics* the most common approach is to avoid moving explicitly hypotheses back and forth between the context and the conclusion of the goal. On the contrary in SSReflect all bookkeeping is performed on the conclusion of the goal, using for that purpose a couple of syntactic constructions behaving similar to tacticals (and often named as such in this chapter). The `:` tactical moves hypotheses from the context to the conclusion, while `=>` moves hypotheses from the conclusion to the context, and `in` moves back and forth a hypothesis from the context to the conclusion for the time of applying an action to it.

While naming hypotheses is commonly done by means of an `as` clause in the basic model of Chapter *Tactics*, it is here to `=>` that this task is devoted. Tactics frequently leave new assumptions in the conclusion, and are often followed by `=>` to explicitly name them. While generalizing the goal is normally not explicitly needed in Chapter *Tactics*, it is an explicit operation performed by `:`.

See also:

Bookkeeping

Beside the difference of bookkeeping model, this chapter includes specific tactics which have no explicit counterpart in Chapter *Tactics* such as tactics to mix forward steps and generalizations as *`generally have`* or *`without loss`*.

SSReflect adopts the point of view that rewriting, definition expansion and partial evaluation participate all to a same concept of rewriting a goal in a larger sense. As such, all these functionalities are provided by the *`rewrite`* tactic.

SSReflect includes a little language of patterns to select subterms in tactics or tacticals where it matters. Its most notable application is in the *`rewrite`* tactic, where patterns are used to specify where the rewriting step has to take place.

Finally, SSReflect supports so-called reflection steps, typically allowing to switch back and forth between the computational view and logical view of a concept.

To conclude it is worth mentioning that SSReflect tactics can be mixed with non SSReflect tactics in the same proof, or in the same Ltac expression. The few exceptions to this statement are described in section *Compatibility issues*.

Acknowledgments

The authors would like to thank Frédéric Blanqui, François Pottier and Laurence Rideau for their comments and suggestions.

5.7.2 Usage

Getting started

To be available, the tactics presented in this manual need the following minimal set of libraries to be loaded: `ssreflect.v`, `ssrfun.v` and `ssrbool.v`. Moreover, these tactics come with a methodology specific to the authors of SSReflect and which requires a few options to be set in a different way than in their default way. All in all, this corresponds to working in the following context:

```
From Coq Require Import ssreflect ssrfun ssrbool.
Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

See also:

Implicit Arguments, *Strict Implicit*, *Printing Implicit Defensive*

Compatibility issues

Requiring the above modules creates an environment which is mostly compatible with the rest of Coq, up to a few discrepancies:

- New keywords (`is`) might clash with variable, constant, tactic or tactical names, or with quasi-keywords in tactic or vernacular notations.
- New tactic(al)s names (*last*, *done*, *have*, *suffices*, *suff*, *without loss*, *wlog*, *congr*, *unlock*) might clash with user tactic names.
- Identifiers with both leading and trailing `_`, such as `_x_`, are reserved by SSReflect and cannot appear in scripts.
- The extensions to the *rewrite* tactic are partly incompatible with those available in current versions of Coq; in particular: `rewrite .. in (type of k)` or `rewrite .. in *` or any other variant of *rewrite* will not work, and the SSReflect syntax and semantics for occurrence selection and rule chaining is different. Use an explicit rewrite direction (`rewrite <- …` or `rewrite -> …`) to access the Coq rewrite tactic.
- New symbols (`//`, `/=`, `//=`) might clash with adjacent existing symbols. This can be avoided by inserting white spaces.
- New constant and theorem names might clash with the user theory. This can be avoided by not importing all of SSReflect:

  ```
  From Coq Require ssreflect.
  Import ssreflect.SsrSyntax.
  ```

  Note that the full syntax of SSReflect's rewrite and reserved identifiers are enabled only if the ssreflect module has been required and if `SsrSyntax` has been imported. Thus a file that requires (without importing) `ssreflect` and imports `SsrSyntax`, can be required and imported without automatically enabling SSReflect's extended rewrite syntax and reserved identifiers.

- Some user notations (in particular, defining an infix `;`) might interfere with the "open term", parenthesis free, syntax of tactics such as have, set and pose.
- The generalization of if statements to non-Boolean conditions is turned off by SSReflect, because it is mostly subsumed by Coercion to `bool` of the `sumXXX` types (declared in `ssrfun.v`) and the `if` *`term`* `is` *`pattern`* `then` *`term`* `else` *`term`* construct (see *Pattern conditional*). To use the generalized form, turn off the SSReflect Boolean `if` notation using the command: `Close Scope boolean_if_scope`.
- The following flags can be unset to make SSReflect more compatible with parts of Coq:

Flag: `SsrRewrite`

Controls whether the incompatible rewrite syntax is enabled (the default). Disabling the flag makes the syntax compatible with other parts of Coq.

Flag: `SsrIdents`

Controls whether tactics can refer to SSReflect-generated variables that are in the form _xxx_. Scripts with explicit references to such variables are fragile; they are prone to failure if the proof is later modified or if the details of variable name generation change in future releases of Coq.

The default is on, which gives an error message when the user tries to create such identifiers. Disabling the flag generates a warning instead, increasing compatibility with other parts of Coq.

5.7.3 Gallina extensions

Small-scale reflection makes an extensive use of the programming subset of Gallina, Coq's logical specification language. This subset is quite suited to the description of functions on representations, because it closely follows the well-established design of the ML programming language. The SSReflect extension provides three additions to Gallina, for pattern assignment, pattern testing, and polymorphism; these mitigate minor but annoying discrepancies between Gallina and ML.

Pattern assignment

The SSReflect extension provides the following construct for irrefutable pattern matching, that is, destructuring assignment:

`term += let:` *`pattern`* **`:=`** *`term`* **`in`** *`term`*

Note the colon `:` after the `let` keyword, which avoids any ambiguity with a function definition or Coq's basic destructuring let. The let: construct differs from the latter in that

- The pattern can be nested (deep pattern matching), in particular, this allows expression of the form:

```
let: exist (x, y) p_xy := Hp in … .
```

- The destructured constructor is explicitly given in the pattern, and is used for type inference.

Example

```
Definition f u := let: (m, n) := u in m + n.
    f is defined

Check f.
    f
         : nat * nat -> nat
```

Using `let:` Coq infers a type for `f`, whereas with a usual `let` the same term requires an extra type annotation in order to type check.

```
Fail Definition f u := let (m, n) := u in m + n.
    The command has indeed failed with message:
    Cannot infer a type for this expression.
```

The `let:` construct is just (more legible) notation for the primitive Gallina expression `match` *term* `with` *pattern* `=>` *term* `end`.

The SSReflect destructuring assignment supports all the dependent match annotations; the full syntax is

`term += let:` *pattern* [`as` *ident*]? [`in` *pattern*]? `:=` *term* [`return` *term*]? `in` *term*

where the second *pattern* and the second *term* are *types*.

When the `as` and `return` keywords are both present, then *ident* is bound in both the second *pattern* and the second *term*; variables in the optional type *pattern* are bound only in the second term, and other variables in the first *pattern* are bound only in the third *term*, however.

Pattern conditional

The following construct can be used for a refutable pattern matching, that is, pattern testing:

`term += if` *term* `is` *pattern* `then` *term* `else` *term*

Although this construct is not strictly ML (it does exist in variants such as the pattern calculus or the ρ-calculus), it turns out to be very convenient for writing functions on representations, because most such functions manipulate simple data types such as Peano integers, options, lists, or binary trees, and the pattern conditional above is almost always the right construct for analyzing such simple types. For example, the null and all list function(al)s can be defined as follows:

Example

```
Variable d: Set.
    d is declared

Fixpoint null (s : list d) :=
  if s is nil then true else false.
    null is defined
    null is recursively defined (decreasing on 1st argument)

Variable a : d -> bool.
    a is declared

Fixpoint all (s : list d) : bool :=
  if s is cons x s' then a x && all s' else true.
    all is defined
    all is recursively defined (decreasing on 1st argument)
```

The pattern conditional also provides a notation for destructuring assignment with a refutable pattern, adapted to the pure functional setting of Gallina, which lacks a `Match_Failure` exception.

Like `let:` above, the `if…is` construct is just (more legible) notation for the primitive Gallina expression `match` *term* `with` *pattern* `=>` *term* `| _ =>` *term* `end`.

Similarly, it will always be displayed as the expansion of this form in terms of primitive match expressions (where the default expression may be replicated).

Explicit pattern testing also largely subsumes the generalization of the `if` construct to all binary data types; compare `if` *term* `is inl _ then` *term* `else` *term* and `if` *term* `then` *term* `else` *term*.

The latter appears to be marginally shorter, but it is quite ambiguous, and indeed often requires an explicit annotation `(term : {_} + {_})` to type check, which evens the character count.

Therefore, SSReflect restricts by default the condition of a plain if construct to the standard `bool` type; this avoids spurious type annotations.

Example

```
Definition orb b1 b2 := if b1 then true else b2.
    orb is defined
```

As pointed out in section *Compatibility issues*, this restriction can be removed with the command:

`Close Scope boolean_if_scope.`

Like `let:` above, the `if-is-then-else` construct supports the dependent match annotations:

`term += if` *term* **`is`** *pattern* **`as`** *ident* **`in`** *pattern* **`return`** *term* **`then`** *term* **`else`** *term*

As in `let:` the variable *ident* (and those in the type pattern) are bound in the second *term*; *ident* is also bound in the third *term* (but not in the fourth *term*), while the variables in the first *pattern* are bound only in the third *term*.

Another variant allows to treat the `else` case first:

`term += if` *term* **`isn't`** *pattern* **`then`** *term* **`else`** *term*

Note that *pattern* eventually binds variables in the third *term* and not in the second *term*.

Parametric polymorphism

Unlike ML, polymorphism in core Gallina is explicit: the type parameters of polymorphic functions must be declared explicitly, and supplied at each point of use. However, Coq provides two features to suppress redundant parameters:

- Sections are used to provide (possibly implicit) parameters for a set of definitions.
- Implicit arguments declarations are used to tell Coq to use type inference to deduce some parameters from the context at each point of call.

The combination of these features provides a fairly good emulation of ML-style polymorphism, but unfortunately this emulation breaks down for higher-order programming. Implicit arguments are indeed not inferred at all points of use, but only at points of call, leading to expressions such as

Example

```
Definition all_null (s : list T) := all (@null T) s.
    all_null is defined
```

Unfortunately, such higher-order expressions are quite frequent in representation functions, especially those which use Coq's `Structures` to emulate Haskell typeclasses.

Therefore, SSReflect provides a variant of Coq's implicit argument declaration, which causes Coq to fill in some implicit parameters at each point of use, e.g., the above definition can be written:

Example

```
Prenex Implicits null.
Definition all_null (s : list T) := all null s.
    all_null is defined
```

Better yet, it can be omitted entirely, since `all_null s` isn't much of an improvement over `all null s`.

The syntax of the new declaration is

Command: `Prenex Implicits` *ident*$_i^+$

This command checks that each *ident*$_i$ is the name of a functional constant, whose implicit arguments are prenex, i.e., the first $n_i > 0$ arguments of *ident*$_i$ are implicit; then it assigns `Maximal Implicit` status to these arguments.

As these prenex implicit arguments are ubiquitous and have often large display strings, it is strongly recommended to change the default display settings of Coq so that they are not printed (except after a `Set Printing All` command). All SSReflect library files thus start with the incantation

```
Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

Anonymous arguments

When in a definition, the type of a certain argument is mandatory, but not its name, one usually uses "arrow" abstractions for prenex arguments, or the `(_ : term)` syntax for inner arguments. In SSReflect, the latter can be replaced by the open syntax `of term` or (equivalently) `& term`, which are both syntactically equivalent to a `(_ : term)` expression. This feature almost behaves as the following extension of the binder syntax:

`binder +=` `&` *term* | `of` *term*

Caveat: `& T` and `of T` abbreviations have to appear at the end of a binder list. For instance, the usual two-constructor polymorphic type list, i.e. the one of the standard `List` library, can be defined by the following declaration:

Example

```
Inductive list (A : Type) : Type := nil | cons of A & list A.
    list is defined
    list_rect is defined
    list_ind is defined
    list_rec is defined
    list_sind is defined
```

Wildcards

The terms passed as arguments to SSReflect tactics can contain *holes*, materialized by wildcards `_`. Since SSReflect allows a more powerful form of type inference for these arguments, it enhances the possibilities of using such wildcards. These holes are in particular used as a convenient shorthand for abstractions, especially in local definitions or type expressions.

Wildcards may be interpreted as abstractions (see for example sections *Definitions* and *Structure*), or their content can be inferred from the whole context of the goal (see for example section *Abbreviations*).

Definitions

`pose`

This tactic allows to add a defined constant to a proof context. SSReflect generalizes this tactic in several ways. In particular, the SSReflect pose tactic supports *open syntax*: the body of the definition does not need surrounding parentheses. For instance:

```
pose t := x + y.
```

is a valid tactic expression.

The pose tactic is also improved for the local definition of higher order terms. Local definitions of functions can use the same syntax as global ones. For example, the tactic `pose` supports parameters:

Example

```
Lemma test : True.
    1 subgoal

      ============================
      True

pose f x y := x + y.
    1 subgoal

      f := fun x y : nat => x + y : nat -> nat -> nat
      ============================
      True
```

The SSReflect pose tactic also supports (co)fixpoints, by providing the local counterpart of the `Fixpoint f := …` and `CoFixpoint f := …` constructs. For instance, the following tactic:

```
pose fix f (x y : nat) {struct x} : nat :=
  if x is S p then S (f p y) else 0.
```

defines a local fixpoint `f`, which mimics the standard plus operation on natural numbers.

Similarly, local cofixpoints can be defined by a tactic of the form:

```
pose cofix f (arg : T) := … .
```

The possibility to include wildcards in the body of the definitions offers a smooth way of defining local abstractions. The type of “holes” is guessed by type inference, and the holes are abstracted. For instance the tactic:

```
pose f := _ + 1.
```

is shorthand for:

```
pose f n := n + 1.
```

When the local definition of a function involves both arguments and holes, hole abstractions appear first. For instance, the tactic:

```
pose f x := x + _.
```

is shorthand for:

```
pose f n x := x + n.
```

The interaction of the pose tactic with the interpretation of implicit arguments results in a powerful and concise syntax for local definitions involving dependent types. For instance, the tactic:

```
pose f x y := (x, y).
```

adds to the context the local definition:

```
pose f (Tx Ty : Type) (x : Tx) (y : Ty) := (x, y).
```

The generalization of wildcards makes the use of the pose tactic resemble ML-like definitions of polymorphic functions.

Abbreviations

set *ident* [: *term*]? := [*occ_switch*]? *term*

The SSReflect **set** tactic performs abbreviations: it introduces a defined constant for a subterm appearing in the goal and/or in the context.

SSReflect extends the *set* tactic by supplying:

- an open syntax, similarly to the *pose (ssreflect)* tactic;
- a more aggressive matching algorithm;
- an improved interpretation of wildcards, taking advantage of the matching algorithm;
- an improved occurrence selection mechanism allowing to abstract only selected occurrences of a term.

occ_switch ::= { [+ | -]? *num** **}**

where:

- *ident* is a fresh identifier chosen by the user.
- term 1 is an optional type annotation. The type annotation term 1 can be given in open syntax (no surrounding parentheses). If no *occ_switch* (described hereafter) is present, it is also the case for the second *term*. On the other hand, in presence of *occ_switch*, parentheses surrounding the second *term* are mandatory.
- In the occurrence switch *occ_switch*, if the first element of the list is a natural, this element should be a number, and not an Ltac variable. The empty list {} is not interpreted as a valid occurrence switch, it is rather used as a flag to signal the intent of the user to clear the name following it (see *Occurrence switches and redex switches* and *Introduction in the context*)

The tactic:

Example

```
Lemma test x :  f x + f x = f x.
    1 subgoal

      x : nat
      ============================
      f x + f x = f x

set t := f _.
    1 subgoal

      x : nat
      t := f x : nat
      ============================
      t + t = t

set t := {2}(f _).
    1 subgoal

      x : nat
      t := f x : nat
      ============================
      f x + t = f x
```

The type annotation may contain wildcards, which will be filled with the appropriate value by the matching process.

The tactic first tries to find a subterm of the goal matching the second *term* (and its type), and stops at the first subterm it finds. Then the occurrences of this subterm selected by the optional *occ_switch* are replaced by *ident* and a definition *ident* := *term* is added to the context. If no *occ_switch* is present, then all the occurrences are abstracted.

Matching

The matching algorithm compares a pattern *term* with a subterm of the goal by comparing their heads and then pairwise unifying their arguments (modulo conversion). Head symbols match under the following conditions:

- If the head of *term* is a constant, then it should be syntactically equal to the head symbol of the subterm.
- If this head is a projection of a canonical structure, then canonical structure equations are used for the matching.
- If the head of term is *not* a constant, the subterm should have the same structure (λ abstraction, let…in structure …).
- If the head of *term* is a hole, the subterm should have at least as many arguments as *term*.

Example

```
Lemma test (x y z : nat) :  x + y = z.
    1 subgoal

      x, y, z : nat
      ============================
```

(continues on next page)

(continued from previous page)

```
      x + y = z

set t := _ x.
    1 subgoal

      x, y, z : nat
      t := Nat.add x : nat -> nat
      ============================
      t y = z
```

- In the special case where `term` is of the form `(let f := t0 in f) t1 … tn` , then the pattern `term` is treated as `(_ t1 … tn)`. For each subterm in the goal having the form `(A u1 … um)` with m n, the matching algorithm successively tries to find the largest partial application `(A u1 … uj)` convertible to the head `t0` of `term`.

  Example

  ```
  Lemma test : (let f x y z := x + y + z in f 1) 2 3 = 6.
      1 subgoal

        ============================
        (let f := fun x y z : nat => x + y + z in f 1) 2 3 = 6

  set t := (let g y z := S y + z in g) 2.
      1 subgoal

        t := unkeyed (fun y z : nat => S y + z) 2 : nat -> nat
        ============================
        t 3 = 6
  ```

  The notation `unkeyed` defined in `ssreflect.v` is a shorthand for the degenerate term `let x := … in x`.

Moreover:

- Multiple holes in `term` are treated as independent placeholders.

  Example

  ```
  Lemma test x y z : x + y = z.
      1 subgoal

        x, y, z : nat
        ============================
        x + y = z

  set t := _ + _.
      1 subgoal

        x, y, z : nat
        t := x + y : nat
        ============================
        t = z
  ```

- The type of the subterm matched should fit the type (possibly casted by some type annotations) of the pattern `term`.
- The replacement of the subterm found by the instantiated pattern should not capture variables. In the example above `x` is bound and should not be captured.

Example

```
Lemma test : forall x : nat, x + 1 = 0.
    1 subgoal

      ============================
      forall x : nat, x + 1 = 0

Fail set t := _ + 1.
    The command has indeed failed with message:
    The pattern (_ + 1) did not match and has holes. Did you mean pose?
```

- Typeclass inference should fill in any residual hole, but matching should never assign a value to a global existential variable.

Occurrence selection

SSReflect provides a generic syntax for the selection of occurrences by their position indexes. These *occurrence switches* are shared by all SSReflect tactics which require control on subterm selection like rewriting, generalization, …

An *occurrence switch* can be:

- A list natural numbers `{+ n1 … nm}` of occurrences affected by the tactic.

Example

```
Lemma test : f 2 + f 8 = f 2 + f 2.
    1 subgoal

      ============================
      f 2 + f 8 = f 2 + f 2

set x := {+1 3}(f 2).
    1 subgoal

      x := f 2 : nat
      ============================
      x + f 8 = f 2 + x
```

Notice that some occurrences of a given term may be hidden to the user, for example because of a notation. The vernacular `Set Printing All` command displays all these hidden occurrences and should be used to find the correct coding of the occurrences to be selected[298].

Example

[298] Unfortunately, even after a call to the Set Printing All command, some occurrences are still not displayed to the user, essentially the ones possibly hidden in the predicate of a dependent match structure.

```
Notation "a < b":= (le (S a) b).
Lemma test x y : x < y -> S x < S y.
    1 subgoal

      x, y : nat
      ============================
      x < y -> S x < S y

set t := S x.
    1 subgoal

      x, y : nat
      t := S x : nat
      ============================
      t <= y -> t < S y
```

- A list of natural numbers between `{n1 … nm}`. This is equivalent to the previous `{+ n1 … nm}` but the list should start with a number, and not with an Ltac variable.
- A list `{- n1 … nm}` of occurrences *not* to be affected by the tactic.

Example

```
Lemma test : f 2 + f 8 = f 2 + f 2.
    1 subgoal

      ============================
      f 2 + f 8 = f 2 + f 2

set x := {-2}(f 2).
    1 subgoal

      x := f 2 : nat
      ============================
      x + f 8 = f 2 + x
```

Note that, in this goal, it behaves like `set x := {1 3}(f 2).`

- In particular, the switch `{+}` selects *all* the occurrences. This switch is useful to turn off the default behavior of a tactic which automatically clears some assumptions (see section *Discharge* for instance).
- The switch `{-}` imposes that *no* occurrences of the term should be affected by the tactic. The tactic: `set x := {-}(f 2).` leaves the goal unchanged and adds the definition `x := f 2` to the context. This kind of tactic may be used to take advantage of the power of the matching algorithm in a local definition, instead of copying large terms by hand.

It is important to remember that matching *precedes* occurrence selection.

Example

```
Lemma test x y z : x + y = x + y + z.
    1 subgoal

      x, y, z : nat
      ============================
```

(continues on next page)

(continued from previous page)

```
      x + y = x + y + z

set a := {2}(_ + _).
    1 subgoal

      x, y, z : nat
      a := x + y : nat
      ============================
      x + y = a + z
```

Hence, in the following goal, the same tactic fails since there is only one occurrence of the selected term.

Example

```
Lemma test x y z : (x + y) + (z + z) = z + z.
    1 subgoal

      x, y, z : nat
      ============================
      x + y + (z + z) = z + z

Fail set a := {2}(_ + _).
    The command has indeed failed with message:
    Only 1 < 2 occurrence of (x + y + (z + z))
```

Basic localization

It is possible to define an abbreviation for a term appearing in the context of a goal thanks to the `in` tactical.

Variant: `set` *ident* `:=` *term* `in` *ident*+

This variant of *set (ssreflect)* introduces a defined constant called *ident* in the context, and folds it in the context entries mentioned on the right hand side of `in`. The body of *ident* is the first subterm matching these context entries (taken in the given order).

Example

```
Lemma test x t (Hx : x = 3) : x + t = 4.
    1 subgoal

      x, t : nat
      Hx : x = 3
      ============================
      x + t = 4

set z := 3 in Hx.
    1 subgoal

      x, t : nat
      z := 3 : nat
      Hx : x = z
```

(continues on next page)

(continued from previous page)

```
            ============================
            x + t = 4
```

Variant: `set` *`ident`* `:=` *`term`* `in` *`ident`*+ `*`

This variant matches *term* and then folds *ident* similarly in all the given context entries but also folds *ident* in the goal.

Example

```
Lemma test x t (Hx : x = 3) : x + t = 4.
      1 subgoal

        x, t : nat
        Hx : x = 3
        ============================
        x + t = 4

set z := 3 in Hx * .
      1 subgoal

        x, t : nat
        z := 3 : nat
        Hx : x = z
        ============================
        x + t = S z
```

Indeed, remember that 4 is just a notation for (S 3).

The use of the `in` tactical is not limited to the localization of abbreviations: for a complete description of the in tactical, see section *Bookkeeping* and *Localization*.

5.7.4 Basic tactics

A sizable fraction of proof scripts consists of steps that do not ”prove” anything new, but instead perform menial bookkeeping tasks such as selecting the names of constants and assumptions or splitting conjuncts. Although they are logically trivial, bookkeeping steps are extremely important because they define the structure of the data-flow of a proof script. This is especially true for reflection-based proofs, which often involve large numbers of constants and assumptions. Good bookkeeping consists in always explicitly declaring (i.e., naming) all new constants and assumptions in the script, and systematically pruning irrelevant constants and assumptions in the context. This is essential in the context of an interactive development environment (IDE), because it facilitates navigating the proof, allowing to instantly ”jump back” to the point at which a questionable assumption was added, and to find relevant assumptions by browsing the pruned context. While novice or casual Coq users may find the automatic name selection feature convenient, the usage of such a feature severely undermines the readability and maintainability of proof scripts, much like automatic variable declaration in programming languages. The SSReflect tactics are therefore designed to support precise bookkeeping and to eliminate name generation heuristics. The bookkeeping features of SSReflect are implemented as tacticals (or pseudo-tacticals), shared across most SSReflect tactics, and thus form the foundation of the SSReflect proof language.

Bookkeeping

During the course of a proof Coq always present the user with a *sequent* whose general form is:

```
ci : Ti
…
dj := ej : Tj
…
Fk : Pk
…
=================
forall (xl : Tl) …,
let ym := bm in … in
Pn -> … -> C
```

The *goal* to be proved appears below the double line; above the line is the *context* of the sequent, a set of declarations of *constants* `ci` , *defined constants* `dj` , and *facts* `Fk` that can be used to prove the goal (usually, `Ti` , `Tj : Type` and `Pk : Prop`). The various kinds of declarations can come in any order. The top part of the context consists of declarations produced by the Section commands `Variable`, `Let`, and `Hypothesis`. This *section context* is never affected by the SSReflect tactics: they only operate on the lower part — the *proof context*. As in the figure above, the goal often decomposes into a series of (universally) quantified *variables* `(xl : Tl)`, local *definitions* `let ym := bm in`, and *assumptions* `P n ->`, and a *conclusion* `C` (as in the context, variables, definitions, and assumptions can appear in any order). The conclusion is what actually needs to be proved — the rest of the goal can be seen as a part of the proof context that happens to be "below the line".

However, although they are logically equivalent, there are fundamental differences between constants and facts on the one hand, and variables and assumptions on the others. Constants and facts are *unordered*, but *named* explicitly in the proof text; variables and assumptions are *ordered*, but *unnamed*: the display names of variables may change at any time because of α-conversion.

Similarly, basic deductive steps such as apply can only operate on the goal because the Gallina terms that control their action (e.g., the type of the lemma used by `apply`) only provide unnamed bound variables.[299] Since the proof script can only refer directly to the context, it must constantly shift declarations from the goal to the context and conversely in between deductive steps.

In SSReflect these moves are performed by two *tacticals* `=>` and `:`, so that the bookkeeping required by a deductive step can be directly associated to that step, and that tactics in an SSReflect script correspond to actual logical steps in the proof rather than merely shuffle facts. Still, some isolated bookkeeping is unavoidable, such as naming variables and assumptions at the beginning of a proof. SSReflect provides a specific `move` tactic for this purpose.

Now `move` does essentially nothing: it is mostly a placeholder for `=>` and `:`. The `=>` tactical moves variables, local definitions, and assumptions to the context, while the `:` tactical moves facts and constants to the goal.

Example

For example, the proof of[300]

```
Lemma subnK : forall m n, n <= m -> m - n + n = m.
    1 subgoal

      ============================
      forall m n : nat, n <= m -> m - n + n = m
```

[299] Thus scripts that depend on bound variable names, e.g., via intros or with, are inherently fragile.

[300] The name `subnK` reads as "right cancellation rule for nat subtraction".

might start with

```
move=> m n le_n_m.
    1 subgoal

      m, n : nat
      le_n_m : n <= m
      ============================
      m - n + n = m
```

where move does nothing, but `=> m n le_m_n` changes the variables and assumption of the goal in the constants `m n : nat` and the fact `le_n_m : n <= m`, thus exposing the conclusion `m - n + n = m`.

The `:` tactical is the converse of `=>`, indeed it removes facts and constants from the context by turning them into variables and assumptions.

```
move: m le_n_m.
    1 subgoal

      n : nat
      ============================
      forall m : nat, n <= m -> m - n + n = m
```

turns back `m` and `le_m_n` into a variable and an assumption, removing them from the proof context, and changing the goal to `forall m, n <= m -> m - n + n = m` which can be proved by induction on `n` using `elim: n`.

Because they are tacticals, `:` and `=>` can be combined, as in

```
move: m le_n_m => p le_n_p.
```

simultaneously renames `m` and `le_m_n` into `p` and `le_n_p`, respectively, by first turning them into unnamed variables, then turning these variables back into constants and facts.

Furthermore, SSReflect redefines the basic Coq tactics `case`, `elim`, and `apply` so that they can take better advantage of `:` and `=>`. In there SSReflect variants, these tactic operate on the first variable or constant of the goal and they do not use or change the proof context. The `:` tactical is used to operate on an element in the context.

Example

> For instance the proof of `subnK` could continue with `elim: n`. Instead of `elim n` (note, no colon), this has the advantage of removing n from the context. Better yet, this `elim` can be combined with previous move and with the branching version of the `=>` tactical (described in *Introduction in the context*), to encapsulate the inductive step in a single command:

```
Lemma subnK : forall m n, n <= m -> m - n + n = m.
    1 subgoal

      ============================
      forall m n : nat, n <= m -> m - n + n = m

move=> m n le_n_m.
    1 subgoal

      m, n : nat
      le_n_m : n <= m
```

(continues on next page)

(continued from previous page)

```
      ============================
      m - n + n = m

elim: n m le_n_m => [|n IHn] m => [_ | lt_n_m].
    2 subgoals

      m : nat
      ============================
      m - 0 + 0 = m

    subgoal 2 is:
     m - S n + S n = m
```

which breaks down the proof into two subgoals, the second one having in its context `lt_n_m : S n <= m` and `IHn : forall m, n <= m -> m - n + n = m`.

The `:` and `=>` tacticals can be explained very simply if one views the goal as a stack of variables and assumptions piled on a conclusion:

- `tactic : a b c` pushes the context constants `a`, `b`, `c` as goal variables *before* performing tactic.
- `tactic => a b c` pops the top three goal variables as context constants `a`, `b`, `c`, *after* tactic has been performed.

These pushes and pops do not need to balance out as in the examples above, so `move: m le_n_m => p` would rename `m` into `p`, but leave an extra assumption `n <= p` in the goal.

Basic tactics like apply and elim can also be used without the ':' tactical: for example we can directly start a proof of `subnK` by induction on the top variable `m` with

```
elim=> [|m IHm] n le_n.
```

The general form of the localization tactical in is also best explained in terms of the goal stack:

```
tactic in a H1 H2 *.
```

is basically equivalent to

```
move: a H1 H2; tactic => a H1 H2.
```

with two differences: the in tactical will preserve the body of an if a is a defined constant, and if the `*` is omitted it will use a temporary abbreviation to hide the statement of the goal from `tactic`.

The general form of the in tactical can be used directly with the `move`, `case` and `elim` tactics, so that one can write

```
elim: n => [|n IHn] in m le_n_m *.
```

instead of

```
elim: n m le_n_m => [|n IHn] m le_n_m.
```

This is quite useful for inductive proofs that involve many facts.

See section *Localization* for the general syntax and presentation of the in tactical.

The defective tactics

In this section we briefly present the three basic tactics performing context manipulations and the main backward chaining tool.

The move tactic.

`move`

This tactic, in its defective form, behaves like the *hnf* tactic.

Example

```
Require Import ssreflect.
Goal not False.
    1 subgoal

      ============================
      ~ False

move.
    1 subgoal

      ============================
      False -> False
```

More precisely, the *move* tactic inspects the goal and does nothing (*idtac*) if an introduction step is possible, i.e. if the goal is a product or a `let … in`, and performs *hnf* otherwise.

Of course this tactic is most often used in combination with the bookkeeping tacticals (see section *Introduction in the context* and *Discharge*). These combinations mostly subsume the *intros*, *generalize*, *revert*, *rename*, *clear* and *pattern* tactics.

The case tactic

`case`

This tactic performs *primitive case analysis* on (co)inductive types; specifically, it destructs the top variable or assumption of the goal, exposing its constructor(s) and its arguments, as well as setting the value of its type family indices if it belongs to a type family (see section *Type families*).

The SSReflect case tactic has a special behavior on equalities. If the top assumption of the goal is an equality, the case tactic "destructs" it as a set of equalities between the constructor arguments of its left and right hand sides, as per the tactic injection. For example, `case` changes the goal:

```
(x, y) = (1, 2) -> G.
```

into:

```
x = 1 -> y = 2 -> G.
```

Note also that the case of SSReflect performs `False` elimination, even if no branch is generated by this case operation. Hence the tactic *case* on a goal of the form `False -> G` will succeed and prove the goal.

The elim tactic

elim

This tactic performs inductive elimination on inductive types. In its defective form, the tactic performs inductive elimination on a goal whose top assumption has an inductive type.

Example

```
Lemma test m : forall n : nat, m <= n.
    1 subgoal

      m : nat
      ============================
      forall n : nat, m <= n

elim.
    2 subgoals

      m : nat
      ============================
      m <= 0

    subgoal 2 is:
     forall n : nat, m <= n -> m <= S n
```

The apply tactic

apply *term*?

This is the main backward chaining tactic of the proof system. It takes as argument any *term* and applies it to the goal. Assumptions in the type of *term* that don't directly match the goal may generate one or more subgoals.

In its defective form, this tactic is a synonym for:

```
intro top; first [refine top | refine (top _) | refine (top _ _) | …]; clear top.
```

where top is a fresh name, and the sequence of *refine* tactics tries to catch the appropriate number of wildcards to be inserted. Note that this use of the *refine* tactic implies that the tactic tries to match the goal up to expansion of constants and evaluation of subterms.

apply (ssreflect) has a special behavior on goals containing existential metavariables of sort Prop.

Example

```
Lemma test : forall y, 1 < y -> y < 2 -> exists x : { n | n < 3 }, 0 < proj1_sig x.
    1 subgoal

      ============================
      forall y : nat,
      1 < y -> y < 2 -> exists x : {n : nat | n < 3}, 0 < proj1_sig x

move=> y y_gt1 y_lt2; apply: (ex_intro _ (exist _ y _)).
    2 focused subgoals
```

(continues on next page)

(continued from previous page)

```
    (shelved: 2)

      y : nat
      y_gt1 : 1 < y
      y_lt2 : y < 2
      ============================
      y < 3

    subgoal 2 is:
     forall Hyp0 : y < 3, 0 < proj1_sig (exist (fun n : nat => n < 3) y Hyp0)

by apply: lt_trans y_lt2 _.
    1 focused subgoal
    (shelved: 1)

      y : nat
      y_gt1 : 1 < y
      y_lt2 : y < 2
      ============================
      forall Hyp0 : y < 3, 0 < proj1_sig (exist (fun n : nat => n < 3) y Hyp0)

by move=> y_lt3; apply: lt_trans y_gt1.
    No more subgoals.
```

Note that the last `_` of the tactic `apply: (ex_intro _ (exist _ y _))` represents a proof that `y < 3`. Instead of generating the goal:

```
0 < proj1_sig (exist (fun n : nat => n < 3) y ?Goal).
```

the system tries to prove `y < 3` calling the trivial tactic. If it succeeds, let's say because the context contains `H : y < 3`, then the system generates the following goal:

```
0 < proj1_sig (exist (fun n => n < 3) y H).
```

Otherwise the missing proof is considered to be irrelevant, and is thus discharged generating the two goals shown above.

Last, the user can replace the trivial tactic by defining an Ltac expression named `ssrautoprop`.

Discharge

The general syntax of the discharging tactical `:` is:

tactic *ident*? `:` *d_item*+ *clear_switch*?

`d_item ::=` [*occ_switch* | *clear_switch*]? *term*

`clear_switch ::= {` *ident*+ `}`

with the following requirements:

- `tactic` must be one of the four basic tactics described in *The defective tactics*, i.e., `move`, `case`, `elim` or `apply`, the `exact` tactic (section *Terminators*), the `congr` tactic (section *Congruence*), or the application of the *view* tactical '/' (section *Interpreting assumptions*) to one of move, case, or elim.

- The optional *ident* specifies *equation generation* (section *Generation of equations*), and is only allowed if tactic is `move`, `case` or `elim`, or the application of the view tactical '/' (section *Interpreting assumptions*) to `case` or `elim`.
- An *occ_switch* selects occurrences of *term*, as in *Abbreviations*; *occ_switch* is not allowed if `tactic` is `apply` or `exact`.
- A clear item *clear_switch* specifies facts and constants to be deleted from the proof context (as per the clear tactic).

The `:` tactical first *discharges* all the *d_item*, right to left, and then performs tactic, i.e., for each *d_item*, starting with the last one :

1. The SSReflect matching algorithm described in section *Abbreviations* is used to find occurrences of term in the goal, after filling any holes '_' in term; however if tactic is apply or exact a different matching algorithm, described below, is used[301].
2. These occurrences are replaced by a new variable; in particular, if term is a fact, this adds an assumption to the goal.
3. If term is *exactly* the name of a constant or fact in the proof context, it is deleted from the context, unless there is an *occ_switch*.

Finally, tactic is performed just after the first *d_item* has been generalized — that is, between steps 2 and 3. The names listed in the final *clear_switch* (if it is present) are cleared first, before *d_item* n is discharged.

Switches affect the discharging of a *d_item* as follows:

- An *occ_switch* restricts generalization (step 2) to a specific subset of the occurrences of term, as per section *Abbreviations*, and prevents clearing (step 3).
- All the names specified by a *clear_switch* are deleted from the context in step 3, possibly in addition to term.

For example, the tactic:

```
move: n {2}n (refl_equal n).
```

- first generalizes `(refl_equal n : n = n)`;
- then generalizes the second occurrence of `n`.
- finally generalizes all the other occurrences of `n`, and clears `n` from the proof context (assuming n is a proof constant).

Therefore this tactic changes any goal `G` into

```
forall n n0 : nat, n = n0 -> G.
```

where the name `n0` is picked by the Coq display function, and assuming `n` appeared only in `G`.

Finally, note that a discharge operation generalizes defined constants as variables, and not as local definitions. To override this behavior, prefix the name of the local definition with a `@`, like in `move: @n`.

This is in contrast with the behavior of the in tactical (see section *Localization*), which preserves local definitions by default.

[301] Also, a slightly different variant may be used for the first *d_item* of case and elim; see section *Type families*.

Clear rules

The clear step will fail if term is a proof constant that appears in other facts; in that case either the facts should be cleared explicitly with a *clear_switch*, or the clear step should be disabled. The latter can be done by adding an *occ_switch* or simply by putting parentheses around term: both `move: (n).` and `move: {+}n.` generalize `n` without clearing `n` from the proof context.

The clear step will also fail if the *clear_switch* contains a *ident* that is not in the *proof* context. Note that SSReflect never clears a section constant.

If tactic is `move` or `case` and an equation *ident* is given, then clear (step 3) for *d_item* is suppressed (see section *Generation of equations*).

Intro patterns (see section *Introduction in the context*) and the `rewrite` tactic (see section *Rewriting*) let one place a *clear_switch* in the middle of other items (namely identifiers, views and rewrite rules). This can trigger the addition of proof context items to the ones being explicitly cleared, and in turn this can result in clear errors (e.g. if the context item automatically added occurs in the goal). The relevant sections describe ways to avoid the unintended clear of context items.

Matching for apply and exact

The matching algorithm for *d_item* of the SSReflect `apply` and `exact` tactics exploits the type of the first *d_item* to interpret wildcards in the other *d_item* and to determine which occurrences of these should be generalized. Therefore, occur switches are not needed for apply and exact.

Indeed, the SSReflect tactic `apply: H x` is equivalent to `refine (@H _ … _ x); clear H x` with an appropriate number of wildcards between `H` and `x`.

Note that this means that matching for `apply` and `exact` has much more context to interpret wildcards; in particular it can accommodate the `_` *d_item*, which would always be rejected after `move:`.

Example

```
Lemma test (Hfg : forall x, f x = g x) a b : f a = g b.
    1 subgoal

      Hfg : forall x : nat, f x = g x
      a, b : nat
      ============================
      f a = g b

apply: trans_equal (Hfg _) _.
    1 focused subgoal
    (shelved: 1)

      Hfg : forall x : nat, f x = g x
      a, b : nat
      ============================
      g a = g b
```

This tactic is equivalent (see section *Bookkeeping*) to: `refine (trans_equal (Hfg _) _).` and this is a common idiom for applying transitivity on the left hand side of an equation.

The abstract tactic

abstract: *d_item*+

This tactic assigns an abstract constant previously introduced with the `[: ident ]` intro pattern (see section *Introduction in the context*).

In a goal like the following:

```
m : nat
abs : <hidden>
n : nat
=============
m < 5 + n
```

The tactic `abstract: abs n` first generalizes the goal with respect to `n` (that is not visible to the abstract constant abs) and then assigns abs. The resulting goal is:

```
m : nat
n : nat
=============
m < 5 + n
```

Once this subgoal is closed, all other goals having abs in their context see the type assigned to `abs`. In this case:

```
m : nat
abs : forall n, m < 5 + n
=============
…
```

For a more detailed example the reader should refer to section *Structure*.

Introduction in the context

The application of a tactic to a given goal can generate (quantified) variables, assumptions, or definitions, which the user may want to *introduce* as new facts, constants or defined constants, respectively. If the tactic splits the goal into several subgoals, each of them may require the introduction of different constants and facts. Furthermore it is very common to immediately decompose or rewrite with an assumption instead of adding it to the context, as the goal can often be simplified and even proved after this.

All these operations are performed by the introduction tactical `=>`, whose general syntax is

tactic `=>` *i_item*+

`i_item ::=` *i_pattern* | *s_item* | *clear_switch* | *i_view* | *i_block*

`s_item ::=` `/=` | `//` | `//=`

`i_view ::=` `{}`? `/`*term* | `/ltac:(` *tactic* `)`

`i_pattern ::=` *ident* | `>` | `_` | `?` | `*` | `+` | *occ_switch*? `->` | `<-` | `[` *i_item*? \| `]` | `-` | `[:` *iden*

`i_block ::=` `[^` *ident* `]` | `[^~` *ident* | *num* `]`

The => tactical first executes tactic, then the *i_item*s, left to right. An *s_item* specifies a simplification operation; a *clear_switch* specifies context pruning as in *Discharge*. The *i_pattern*s can be seen as a variant of *intro patterns* (see *intros*:) each performs an introduction operation, i.e., pops some variables or assumptions from the goal.

Simplification items

An *s_item* can simplify the set of subgoals or the subgoals themselves:

- // removes all the "trivial" subgoals that can be resolved by the SSReflect tactic *done* described in *Terminators*, i.e., it executes try done.
- /= simplifies the goal by performing partial evaluation, as per the tactic *simpl*[302].
- //= combines both kinds of simplification; it is equivalent to /= //, i.e., simpl; try done.

When an *s_item* immediately precedes a *clear_switch*, then the *clear_switch* is executed *after* the *s_item*, e.g., {IHn}// will solve some subgoals, possibly using the fact IHn, and will erase IHn from the context of the remaining subgoals.

Views

The first entry in the *i_view* grammar rule, /*term*, represents a view (see section *Views and reflection*). It interprets the top of the stack with the view *term*. It is equivalent to move/*term*.

A *clear_switch* that immediately precedes an *i_view* is complemented with the name of the view if an only if the *i_view* is a simple proof context entry[307]. E.g. {}/v is equivalent to /v{v}. This behavior can be avoided by separating the *clear_switch* from the *i_view* with the - intro pattern or by putting parentheses around the view.

A *clear_switch* that immediately precedes an *i_view* is executed after the view application.

If the next *i_item* is a view, then the view is applied to the assumption in top position once all the previous *i_item* have been performed.

The second entry in the *i_view* grammar rule, /ltac:(tactic), executes tactic. Notations can be used to name tactics, for example

```
Notation "'myop'" := (ltac:(my ltac code)) : ssripat_scope.
```

lets one write just /myop in the intro pattern. Note the scope annotation: views are interpreted opening the ssripat scope.

Intro patterns

SSReflect supports the following *i_pattern*s:

ident pops the top variable, assumption, or local definition into a new constant, fact, or defined constant *ident*, respectively. Note that defined constants cannot be introduced when δ-expansion is required to expose the top variable or assumption. A *clear_switch* (even an empty one) immediately preceding an *ident* is complemented with that *ident* if and only if the identifier is a simple proof context entry[307]. As a consequence by prefixing the *ident* with {} one can *replace* a context entry. This behavior can be avoided by separating the *clear_switch* from the *ident* with the - intro pattern.

[302] Except /= does not expand the local definitions created by the SSReflect in tactical.

[307] A simple proof context entry is a naked identifier (i.e. not between parentheses) designating a context entry that is not a section variable.

`>` pops every variable occurring in the rest of the stack. Type class instances are popped even if they don't occur in the rest of the stack. The tactic `move=> >` is equivalent to `move=> ? ?` on a goal such as:

```
forall x y, x < y -> G
```

A typical use if `move=>> H` to name `H` the first assumption, in the example above `x < y`.

`?` pops the top variable into an anonymous constant or fact, whose name is picked by the tactic interpreter. SSReflect only generates names that cannot appear later in the user script[303].

`_` pops the top variable into an anonymous constant that will be deleted from the proof context of all the subgoals produced by the `=>` tactical. They should thus never be displayed, except in an error message if the constant is still actually used in the goal or context after the last *i_item* has been executed (*s_item* can erase goals or terms where the constant appears).

`*` pops all the remaining apparent variables/assumptions as anonymous constants/facts. Unlike `?` and `move` the `*` *i_item* does not expand definitions in the goal to expose quantifiers, so it may be useful to repeat a `move=> *` tactic, e.g., on the goal:

```
forall a b : bool, a <> b
```

a first `move=> *` adds only `_a_ : bool` and `_b_ : bool` to the context; it takes a second `move=> *` to add `_Hyp_ : _a_ = _b_`.

`+` temporarily introduces the top variable. It is discharged at the end of the intro pattern. For example `move=> + y` on a goal:

```
forall x y, P
```

is equivalent to `move=> _x_ y; move: _x_` that results in the goal:

```
forall x, P
```

`occ_switch` ? `->` (resp. *occ_switch* `<-`) pops the top assumption (which should be a rewritable proposition) into an anonymous fact, rewrites (resp. rewrites right to left) the goal with this fact (using the SSReflect `rewrite` tactic described in section *Rewriting*, and honoring the optional occurrence selector), and finally deletes the anonymous fact from the context.

`[` *i_item* * `|` … `|` *i_item* * `]` when it is the very *first* *i_pattern* after tactic `=>` tactical *and* tactic is not a move, is a *branching**i_pattern*. It executes the sequence *i_item*$_i$ on the i-th subgoal produced by tactic. The execution of tactic should thus generate exactly m subgoals, unless the `[…]` *i_pattern* comes after an initial `//` or `//=` *s_item* that closes some of the goals produced by `tactic`, in which case exactly m subgoals should remain after the *s_item*, or we have the trivial branching *i_pattern* [], which always does nothing, regardless of the number of remaining subgoals.

`[` *i_item* * `|` … `|` *i_item* * `]` when it is *not* the first *i_pattern* or when tactic is a `move`, is a *destructing* *i_pattern*. It starts by destructing the top variable, using the SSReflect `case` tactic described in *The defective tactics*. It then behaves as the corresponding branching *i_pattern*, executing the sequence *i_item*$_i$ in the i-th subgoal generated by the case analysis; unless we have the trivial destructing *i_pattern* `[]`, the latter should generate exactly m subgoals, i.e., the top variable should have an inductive type with exactly m constructors[304]. While it is good style to use the *i_item* i * to pop the variables and assumptions corresponding to each constructor, this is not enforced by SSReflect.

`-` does nothing, but counts as an intro pattern. It can also be used to force the interpretation of `[` *i_item* * `|` … `|` *i_item* * `]` as a case analysis like in `move=> -[H1 H2]`. It can also be used to indicate explicitly the link between a view and a name like in `move=> /eqP-H1`. Last, it can serve as a separator between

[303] SSReflect reserves all identifiers of the form "_x_", which is used for such generated names.

[304] More precisely, it should have a quantified inductive type with a assumptions and m − a constructors.

views. Section *Views and reflection*[306] explains in which respect the tactic `move=> /v1/v2` differs from the tactic `move=> /v1-/v2`.

`[:` *`ident`* `…]` introduces in the context an abstract constant for each *`ident`*. Its type has to be fixed later on by using the `abstract` tactic. Before then the type displayed is `<hidden>`.

Note that SSReflect does not support the syntax `(ipat, …, ipat)` for destructing intro patterns.

Clear switch

Clears are deferred until the end of the intro pattern.

Example

```
Lemma test x y : Nat.leb 0 x = true -> (Nat.leb 0 x) && (Nat.leb y 2) = true.
    1 subgoal

      x, y : nat
      ============================
      Nat.leb 0 x = true -> Nat.leb 0 x && Nat.leb y 2 = true

move=> {x} ->.
    1 subgoal

      y : nat
      ============================
      true && Nat.leb y 2 = true
```

If the cleared names are reused in the same intro pattern, a renaming is performed behind the scenes.

Facts mentioned in a clear switch must be valid names in the proof context (excluding the section context).

Branching and destructuring

The rules for interpreting branching and destructing *`i_pattern`* are motivated by the fact that it would be pointless to have a branching pattern if tactic is a `move`, and in most of the remaining cases tactic is `case` or `elim`, which implies destruction. The rules above imply that:

- `move=> [a b].`
- `case=> [a b].`
- `case=> a b.`

are all equivalent, so which one to use is a matter of style; `move` should be used for casual decomposition, such as splitting a pair, and `case` should be used for actual decompositions, in particular for type families (see *Type families*) and proof by contradiction.

The trivial branching *`i_pattern`* can be used to force the branching interpretation, e.g.:

- `case=> [] [a b] c.`
- `move=> [[a b] c].`
- `case; case=> a b c.`

are all equivalent.

[306] The current state of the proof shall be displayed by the Show Proof command of Coq proof mode.

Block introduction

SSReflect supports the following *i_block*s:

`[^ ident ]` *block destructing* *i_pattern*. It performs a case analysis on the top variable and introduces, in one go, all the variables coming from the case analysis. The names of these variables are obtained by taking the names used in the inductive type declaration and prefixing them with *ident*. If the intro pattern immediately follows a call to `elim` with a custom eliminator (see *Interpreting eliminations*) then the names are taken from the ones used in the type of the eliminator.

Example

```
Record r := { a : nat; b := (a, 3); _ : bool; }.
    r is defined
    a is defined
    b is defined

Lemma test : r -> True.
    1 subgoal

      ============================
      r -> True

Proof.
move => [^ x ].
    1 subgoal

      xa : nat
      xb := (xa, 3) : nat * nat
      _x?_ : bool
      ============================
      True
```

`[^~ ident ]` *block destructing* using *ident* as a suffix.

`[^~ num ]` *block destructing* using *num* as a suffix.

Only a *s_item* is allowed between the elimination tactic and the block destructing.

Generation of equations

The generation of named equations option stores the definition of a new constant as an equation. The tactic:

```
move En: (size l) => n.
```

where `l` is a list, replaces `size l` by `n` in the goal and adds the fact `En : size l = n` to the context. This is quite different from:

```
pose n := (size l).
```

which generates a definition `n := (size l)`. It is not possible to generalize or rewrite such a definition; on the other hand, it is automatically expanded during computation, whereas expanding the equation `En` requires explicit rewriting.

The use of this equation name generation option with a `case` or an `elim` tactic changes the status of the first *i_item*, in order to deal with the possible parameters of the constants introduced.

Example

```
Lemma test (a b :nat) : a <> b.
    1 subgoal

      a, b : nat
      ============================
      a <> b

case E : a => [|n].
    2 subgoals

      a, b : nat
      E : a = 0
      ============================
      0 <> b

    subgoal 2 is:
     S n <> b
```

If the user does not provide a branching *i_item* as first *i_item*, or if the *i_item* does not provide enough names for the arguments of a constructor, then the constants generated are introduced under fresh SSReflect names.

Example

```
Lemma test (a b :nat) : a <> b.
    1 subgoal

      a, b : nat
      ============================
      a <> b

case E : a => H.
    2 subgoals

      a, b : nat
      E : a = 0
      H : 0 = b
      ============================
      False

    subgoal 2 is:
     False

Show 2.
    subgoal 2 is:

      a, b, _n_ : nat
      E : a = S _n_
      H : S _n_ = b
      ============================
      False
```

Combining the generation of named equations mechanism with the *case* tactic strengthens the power of a case analysis. On the other hand, when combined with the *elim* tactic, this feature is mostly useful for debug purposes, to trace the values of decomposed parameters and pinpoint failing branches.

Type families

When the top assumption of a goal has an inductive type, two specific operations are possible: the case analysis performed by the *case* tactic, and the application of an induction principle, performed by the *elim* tactic. When this top assumption has an inductive type, which is moreover an instance of a type family, Coq may need help from the user to specify which occurrences of the parameters of the type should be substituted.

Variant: `case:` *d_item*+ / *d_item*+

Variant: `elim:` *d_item*+ / *d_item*+

A specific `/` switch indicates the type family parameters of the type of a *d_item* immediately following this `/` switch. The *d_item* on the right side of the `/` switch are discharged as described in section *Discharge*. The case analysis or elimination will be done on the type of the top assumption after these discharge operations.

Every *d_item* preceding the `/` is interpreted as arguments of this type, which should be an instance of an inductive type family. These terms are not actually generalized, but rather selected for substitution. Occurrence switches can be used to restrict the substitution. If a term is left completely implicit (e.g. writing just `_`), then a pattern is inferred looking at the type of the top assumption. This allows for the compact syntax:

```
case: {2}_ / eqP.
```

where `_` is interpreted as `(_ == _)` since `eqP T a b : reflect (a = b) (a == b)` and reflect is a type family with one index.

Moreover if the *d_item* list is too short, it is padded with an initial sequence of `_` of the right length.

Example

Here is a small example on lists. We define first a function which adds an element at the end of a given list.

```
Require Import List.
Section LastCases.
Variable A : Type.
    A is declared

Implicit Type l : list A.
Fixpoint add_last a l : list A :=
  match l with
 | nil => a :: nil
 | hd :: tl => hd :: (add_last a tl) end.
    add_last is defined
    add_last is recursively defined (decreasing on 2nd argument)
```

Then we define an inductive predicate for case analysis on lists according to their last element:

```
Inductive last_spec : list A -> Type :=
| LastSeq0 : last_spec nil
| LastAdd s x : last_spec (add_last x s).
    last_spec is defined
```

(continues on next page)

(continued from previous page)

```
    last_spec_rect is defined
    last_spec_ind is defined
    last_spec_rec is defined
    last_spec_sind is defined

Theorem lastP : forall l : list A, last_spec l.
    1 subgoal

      A : Type
      ============================
      forall l : list A, last_spec l

Admitted.
    lastP is declared
```

We are now ready to use lastP in conjunction with case.

```
Lemma test l : (length l) * 2 = length (l ++ l).
    1 subgoal

      A : Type
      l : list A
      ============================
      length l * 2 = length (l ++ l)

case: (lastP l).
    2 subgoals

      A : Type
      l : list A
      ============================
      length nil * 2 = length (nil ++ nil)

    subgoal 2 is:
     forall (s : list A) (x : A),
     length (add_last x s) * 2 = length (add_last x s ++ add_last x s)
```

Applied to the same goal, the tactc `case: l / (lastP l)` generates the same subgoals but `l` has been cleared from both contexts:

```
case: l / (lastP l).
    2 subgoals

      A : Type
      ============================
      length nil * 2 = length (nil ++ nil)

    subgoal 2 is:
     forall (s : list A) (x : A),
     length (add_last x s) * 2 = length (add_last x s ++ add_last x s)
```

Again applied to the same goal:

```
case: {1 3}l / (lastP l).
    2 subgoals

      A : Type
```

(continues on next page)

(continued from previous page)

```
      l : list A
      ============================
      length nil * 2 = length (l ++ nil)

    subgoal 2 is:
     forall (s : list A) (x : A),
     length (add_last x s) * 2 = length (l ++ add_last x s)
```

Note that selected occurrences on the left of the / switch have been substituted with l instead of being affected by the case analysis.

The equation name generation feature combined with a type family / switch generates an equation for the *first* dependent *d_item* specified by the user. Again starting with the above goal, the command:

Example

```
Lemma test l : (length l) * 2 = length (l ++ l).
    1 subgoal

      A : Type
      l : list A
      ============================
      length l * 2 = length (l ++ l)

case E: {1 3}l / (lastP l) => [|s x].
    2 subgoals

      A : Type
      l : list A
      E : l = nil
      ============================
      length nil * 2 = length (l ++ nil)

    subgoal 2 is:
     length (add_last x s) * 2 = length (l ++ add_last x s)

Show 2.
    subgoal 2 is:

      A : Type
      l, s : list A
      x : A
      E : l = add_last x s
      ============================
      length (add_last x s) * 2 = length (l ++ add_last x s)
```

There must be at least one *d_item* to the left of the / switch; this prevents any confusion with the view feature. However, the *d_item* to the right of the / are optional, and if they are omitted the first assumption provides the instance of the type family.

The equation always refers to the first *d_item* in the actual tactic call, before any padding with initial _. Thus, if an inductive type has two family parameters, it is possible to have SSReflect generate an equation for the second one by omitting the pattern for the first; note however that this will fail if the type of the second parameter depends on the value of the first parameter.

5.7.5 Control flow

Indentation and bullets

A linear development of Coq scripts gives little information on the structure of the proof. In addition, replaying a proof after some changes in the statement to be proved will usually not display information to distinguish between the various branches of case analysis for instance.

To help the user in this organization of the proof script at development time, SSReflect provides some bullets to highlight the structure of branching proofs. The available bullets are `-`, `+` and `*`. Combined with tabulation, this lets us highlight four nested levels of branching; the most we have ever needed is three. Indeed, the use of "simpl and closing" switches, of terminators (see above section *Terminators*) and selectors (see section *Selectors*) is powerful enough to avoid most of the time more than two levels of indentation.

Here is a fragment of such a structured script:

```
case E1: (abezoutn _ _) => [[| k1] [| k2]].
- rewrite !muln0 !gexpn0 mulg1 => H1.
  move/eqP: (sym_equal F0); rewrite -H1 orderg1 eqn_mul1.
  by case/andP; move/eqP.
- rewrite muln0 gexpn0 mulg1 => H1.
  have F1: t %| t * S k2.+1 - 1.
    apply: (@dvdn_trans (orderg x)); first by rewrite F0; exact: dvdn_mull.
    rewrite orderg_dvd; apply/eqP; apply: (mulgI x).
    rewrite -{1}(gexpn1 x) mulg1 gexpn_add leq_add_sub //.
    by move: P1; case t.
  rewrite dvdn_subr in F1; last by exact: dvdn_mulr.
  + rewrite H1 F0 -{2}(muln1 (p ^ l)); congr (_ * _).
    by apply/eqP; rewrite -dvdn1.
  + by move: P1; case: (t) => [| [| s1]].
- rewrite muln0 gexpn0 mul1g => H1.
...
```

Terminators

To further structure scripts, SSReflect supplies *terminating* tacticals to explicitly close off tactics. When replaying scripts, we then have the nice property that an error immediately occurs when a closed tactic fails to prove its subgoal.

It is hence recommended practice that the proof of any subgoal should end with a tactic which *fails if it does not solve the current goal*, like *`discriminate`*, *`contradiction`* or *`assumption`*.

In fact, SSReflect provides a generic tactical which turns any tactic into a closing one (similar to *`now`*). Its general syntax is:

`by` *`tactic`*

The Ltac expression `by [`*`tactic`*` | `*`tactic`*` | …]` is equivalent to `do [done | by `*`tactic`*` | by `*`tactic`*` | …]`, which corresponds to the standard Ltac expression `first [done | `*`tactic`*`; done | `*`tactic`*`; done | …]`.

In the script provided as example in section *Indentation and bullets*, the paragraph corresponding to each sub-case ends with a tactic line prefixed with a `by`, like in:

```
by apply/eqP; rewrite -dvdn1.
```

`done`

The *`by`* tactical is implemented using the user-defined, and extensible *`done`* tactic. This *`done`* tactic tries to solve the current goal by some trivial means and fails if it doesn't succeed. Indeed, the tactic expression by *tactic* is equivalent to *tactic*; done.

Conversely, the tactic by [ ] is equivalent to *done*.

The default implementation of the done tactic, in the ssreflect.v file, is:

```
Ltac done :=
  trivial; hnf; intros; solve
   [ do ![solve [trivial | apply: sym_equal; trivial]
         | discriminate | contradiction | split]
   | case not_locked_false_eq_true; assumption
   | match goal with H : ~ _ |- _ => solve [case H; trivial] end ].
```

The lemma not_locked_false_eq_true is needed to discriminate *locked* boolean predicates (see section *Locking, unlocking*). The iterator tactical do is presented in section *Iteration*. This tactic can be customized by the user, for instance to include an *auto* tactic.

A natural and common way of closing a goal is to apply a lemma which is the exact one needed for the goal to be solved. The defective form of the tactic:

```
exact.
```

is equivalent to:

```
do [done | by move=> top; apply top].
```

where top is a fresh name assigned to the top assumption of the goal. This applied form is supported by the : discharge tactical, and the tactic:

```
exact: MyLemma.
```

is equivalent to:

```
by apply: MyLemma.
```

(see section *Discharge* for the documentation of the apply: combination).

Warning: The list of tactics (possibly chained by semicolons) that follows the by keyword is considered to be a parenthesized block applied to the current goal. Hence for example if the tactic:

```
by rewrite my_lemma1.
```

succeeds, then the tactic:

```
by rewrite my_lemma1; apply my_lemma2.
```

usually fails since it is equivalent to:

```
by (rewrite my_lemma1; apply my_lemma2).
```

Selectors

last
first

When composing tactics, the two tacticals first and last let the user restrict the application of a tactic to only one of the subgoals generated by the previous tactic. This covers the frequent cases where a tactic generates two subgoals one of which can be easily disposed of.

This is another powerful way of linearization of scripts, since it happens very often that a trivial subgoal can be solved in a less than one line tactic. For instance, *tactic* `; last by` *tactic* tries to solve the last subgoal generated by the first tactic using the given second tactic, and fails if it does not succeed. Its analogue *tactic* `; first by` *tactic* tries to solve the first subgoal generated by the first tactic using the second given tactic, and fails if it does not succeed.

SSReflect also offers an extension of this facility, by supplying tactics to *permute* the subgoals generated by a tactic.

Variant: `last first`
Variant: `first last`

These two equivalent tactics invert the order of the subgoals in focus.

Variant: `last` *num* `first`

If *num*'s value is k, this tactic rotates the n subgoals G_1 , ..., G_n in focus. The first subgoal becomes G_{n+1-k} and the circular order of subgoals remains unchanged.

`first` *num* `last`

If *num*'s value is k, this tactic rotates the n subgoals G_1 , ..., G_n in focus. The first subgoal becomes G_k and the circular order of subgoals remains unchanged.

Finally, the tactics `last` and `first` combine with the branching syntax of Ltac: if the tactic generates n subgoals on a given goal, then the tactic

```
tactic ; last k [ tactic1 |…| tacticm ] || tacticn.
```

where natural denotes the integer k as above, applies tactic1 to the $n - k + 1$-th goal, … tacticm to the $n - k + 2$-th goal and tacticn to the others.

Example

Here is a small example on lists. We define first a function which adds an element at the end of a given list.

```
Inductive test : nat -> Prop :=
| C1 n of n = 1 : test n
| C2 n of n = 2 : test n
| C3 n of n = 3 : test n
| C4 n of n = 4 : test n.
    test is defined
    test_ind is defined
    test_sind is defined

Lemma example n (t : test n) : True.
    1 subgoal

      n : nat
      t : test n
      ============================
      True

case: t; last 2 [move=> k| move=> l]; idtac.
    4 subgoals

      n : nat
      ============================
      forall n0 : nat, n0 = 1 -> True

    subgoal 2 is:
```

(continues on next page)

(continued from previous page)

```
 k = 2 -> True
subgoal 3 is:
 l = 3 -> True
subgoal 4 is:
 forall n0 : nat, n0 = 4 -> True
```

Iteration

do *num*? *tactic* | [*tactic*+|]

This tactical offers an accurate control on the repetition of tactics. *mult* is a *multiplier*.

Brackets can only be omitted if a single tactic is given *and* a multiplier is present.

A tactic of the form:

```
do [ tactic 1 | … | tactic n ].
```

is equivalent to the standard Ltac expression:

```
first [ tactic 1 | … | tactic n ].
```

The optional multiplier *mult* specifies how many times the action of tactic should be repeated on the current subgoal.

There are four kinds of multipliers:

mult ::= *num* ! | ! | *num* ? | ?

Their meaning is:

- `n!` the step tactic is repeated exactly n times (where n is a positive integer argument).
- `!` the step tactic is repeated as many times as possible, and done at least once.
- `?` the step tactic is repeated as many times as possible, optionally.
- `n?` the step tactic is repeated up to n times, optionally.

For instance, the tactic:

```
tactic; do 1? rewrite mult_comm.
```

rewrites at most one time the lemma `mult_comm` in all the subgoals generated by tactic, whereas the tactic:

```
tactic; do 2! rewrite mult_comm.
```

rewrites exactly two times the lemma `mult_comm` in all the subgoals generated by tactic, and fails if this rewrite is not possible in some subgoal.

Note that the combination of multipliers and rewrite is so often used that multipliers are in fact integrated to the syntax of the SSReflect rewrite tactic, see section *Rewriting*.

Localization

In sections *Basic localization* and *Bookkeeping*, we have already presented the *localization* tactical in, whose general syntax is:

tactic in *ident*+ *?

where *ident* is a name in the context. On the left side of in, tactic can be move, case, elim, rewrite, set, or any tactic formed with the general iteration tactical do (see section *Iteration*).

The operation described by tactic is performed in the facts listed after in and in the goal if a * ends the list of names.

The in tactical successively:

- generalizes the selected hypotheses, possibly "protecting" the goal if * is not present,
- performs tactic, on the obtained goal,
- reintroduces the generalized facts, under the same names.

This defective form of the do tactical is useful to avoid clashes between standard Ltac in and the SSReflect tactical in.

Example

```
Ltac mytac H := rewrite H.
    mytac is defined

Lemma test x y (H1 : x = y) (H2 : y = 3) : x + y = 6.
    1 subgoal

      x, y : nat
      H1 : x = y
      H2 : y = 3
      ============================
      x + y = 6

do [mytac H2] in H1 *.
    1 subgoal

      x, y : nat
      H2 : y = 3
      H1 : x = 3
      ============================
      x + 3 = 6
```

the last tactic rewrites the hypothesis H2 : y = 3 both in H1 : x = y and in the goal x + y = 6.

By default in keeps the body of local definitions. To erase the body of a local definition during the generalization phase, the name of the local definition must be written between parentheses, like in rewrite H in H1 (def_n) H2.

Variant: *tactic* in [*clear_switch* | @? *ident* | (*ident*) | (@? *ident* := *c_pattern*)]+ *?

This is the most general form of the in tactical. In its simplest form the last option lets one rename hypotheses that can't be cleared (like section variables). For example, (y := x) generalizes over x and reintroduces the generalized variable under the name y (and does not clear x). For a more precise description of this form of localization refer to *Advanced generalization*.

Structure

Forward reasoning structures the script by explicitly specifying some assumptions to be added to the proof context. It is closely associated with the declarative style of proof, since an extensive use of these highlighted statements make the script closer to a (very detailed) textbook proof.

Forward chaining tactics allow to state an intermediate lemma and start a piece of script dedicated to the proof of this statement. The use of closing tactics (see section *Terminators*) and of indentation makes syntactically explicit the portion of the script building the proof of the intermediate statement.

The have tactic.

`have : term`

This is the main SSReflect forward reasoning tactic. It can be used in two modes: one starts a new (sub)proof for an intermediate result in the main proof, and the other provides explicitly a proof term for this intermediate step.

This tactic supports open syntax for *term*. Applied to a goal `G`, it generates a first subgoal requiring a proof of *term* in the context of `G`. The second generated subgoal is of the form `term -> G`, where term becomes the new top assumption, instead of being introduced with a fresh name. At the proof-term level, the have tactic creates a β redex, and introduces the lemma under a fresh name, automatically chosen.

Like in the case of the `pose (ssreflect)` tactic (see section *Definitions*), the types of the holes are abstracted in term.

Example

```
Lemma test : True.
    1 subgoal

      ============================
      True

have: _ * 0 = 0.
    2 subgoals

      ============================
      forall n : nat, n * 0 = 0

    subgoal 2 is:
     (forall n : nat, n * 0 = 0) -> True
```

The invocation of `have` is equivalent to:

```
have: forall n : nat, n * 0 = 0.
    2 subgoals

      ============================
      forall n : nat, n * 0 = 0

    subgoal 2 is:
     (forall n : nat, n * 0 = 0) -> True
```

The have tactic also enjoys the same abstraction mechanism as the `pose` tactic for the non-inferred implicit arguments. For instance, the tactic:

Example

```
have: forall x y, (x, y) = (x, y + 0).
    2 subgoals

      ============================
      forall (T : Type) (x : T) (y : nat), (x, y) = (x, y + 0)

    subgoal 2 is:
     (forall (T : Type) (x : T) (y : nat), (x, y) = (x, y + 0)) -> True
```

opens a new subgoal where the type of `x` is quantified.

The behavior of the defective have tactic makes it possible to generalize it in the following general construction:

have *i_item* * *i_pattern* ? (*s_item* | *ssr_binder* +) ? : *term* ? (:= *term* | **by** *tactic*) ?

Open syntax is supported for both *term*. For the description of *i_item* and *s_item* see section *Introduction in the context*. The first mode of the have tactic, which opens a sub-proof for an intermediate result, uses tactics of the form:

Variant: **have** *clear_switch i_item* : *term* **by** *tactic*

which behave like:

```
have: term ; first by tactic.
move=> clear_switch i_item.
```

Note that the *clear_switch precedes* the *i_item*, which allows to reuse a name of the context, possibly used by the proof of the assumption, to introduce the new assumption itself.

The `by` feature is especially convenient when the proof script of the statement is very short, basically when it fits in one line like in:

```
have H23 : 3 + 2 = 2 + 3 by rewrite addnC.
```

The possibility of using *i_item* supplies a very concise syntax for the further use of the intermediate step. For instance,

Example

```
Lemma test a : 3 * a - 1 = a.
    1 subgoal

      a : nat
      ============================
      3 * a - 1 = a

have -> : forall x, x * a = a.
    2 subgoals

      a : nat
```

(continues on next page)

(continued from previous page)

```
      ============================
      forall x : nat, x * a = a

    subgoal 2 is:
     a - 1 = a
```

Note how the second goal was rewritten using the stated equality. Also note that in this last subgoal, the intermediate result does not appear in the context.

Thanks to the deferred execution of clears, the following idiom is also supported (assuming x occurs in the goal only):

```
have {x} -> : x = y.
```

Another frequent use of the intro patterns combined with `have` is the destruction of existential assumptions like in the tactic:

Example

```
Lemma test : True.
    1 subgoal

      ============================
      True

have [x Px]: exists x : nat, x > 0; last first.
    2 subgoals

      x : nat
      Px : x > 0
      ============================
      True

    subgoal 2 is:
     exists x : nat, x > 0
```

An alternative use of the `have` tactic is to provide the explicit proof term for the intermediate lemma, using tactics of the form:

Variant: `have` *ident*? `:=` *term*

This tactic creates a new assumption of type the type of *term*. If the optional *ident* is present, this assumption is introduced under the name *ident*. Note that the body of the constant is lost for the user.

Again, non inferred implicit arguments and explicit holes are abstracted.

Example

```
Lemma test : True.
    1 subgoal

      ============================
      True
```

(continues on next page)

(continued from previous page)

```
have H := forall x, (x, x) = (x, x).
    1 subgoal

      H : Type -> Prop
      ============================
      True
```

adds to the context `H : Type -> Prop.` This is a schematic example but the feature is specially useful when the proof term to give involves for instance a lemma with some hidden implicit arguments.

After the *i_pattern*, a list of binders is allowed.

Example

```
Lemma test : True.
    1 subgoal

      ============================
      True

have H x (y : nat) : 2 * x + y = x + x + y by omega.
    1 subgoal

      H : forall x y : nat, 2 * x + y = x + x + y
      ============================
      True
```

A proof term provided after `:=` can mention these bound variables (that are automatically introduced with the given names). Since the *i_pattern* can be omitted, to avoid ambiguity, bound variables can be surrounded with parentheses even if no type is specified:

```
have (x) : 2 * x = x + x by omega.
    1 subgoal

      ============================
      (forall x : nat, 2 * x = x + x) -> True
```

The *i_item* and *s_item* can be used to interpret the asserted hypothesis with views (see section *Views and reflection*) or simplify the resulting goals.

The *have* tactic also supports a `suff` modifier which allows for asserting that a given statement implies the current goal without copying the goal itself.

Example

```
have suff H : 2 + 2 = 3; last first.
    2 subgoals

      H : 2 + 2 = 3 -> True
      ============================
      True
```

(continues on next page)

(continued from previous page)

```
    subgoal 2 is:
     2 + 2 = 3 -> True
```

Note that H is introduced in the second goal.

The suff modifier is not compatible with the presence of a list of binders.

Generating let in context entries with have

Since SSReflect 1.5 the *have* tactic supports a “transparent” modifier to generate let in context entries: the @ symbol in front of the context entry name.

Example

```
Inductive Ord n := Sub x of x < n.
    Ord is defined
    Ord_rect is defined
    Ord_ind is defined
    Ord_rec is defined
    Ord_sind is defined

Notation "'I_ n" := (Ord n) (at level 8, n at level 2, format "''I_' n").
Arguments Sub {_} _ _.
Lemma test n m (H : m + 1 < n) : True.
    1 subgoal

      n, m : nat
      H : m + 1 < n
      ============================
      True

have @i : 'I_n by apply: (Sub m); omega.
    1 subgoal

      n, m : nat
      H : m + 1 < n
      i := Sub m
             (Decidable.dec_not_not (m < n) (dec_lt m n) (fun ... => ... ...))
       : 'I_n
      ============================
      True
```

Note that the subterm produced by *omega* is in general huge and uninteresting, and hence one may want to hide it. For this purpose the `[: name ]` intro pattern and the tactic `abstract` (see *The abstract tactic*) are provided.

Example

```
Lemma test n m (H : m + 1 < n) : True.
    1 subgoal

      n, m : nat
```

(continues on next page)

(continued from previous page)

```
      H : m + 1 < n
      ============================
      True

have [:pm] @i : 'I_n by apply: (Sub m); abstract: pm; omega.
    1 subgoal

      n, m : nat
      H : m + 1 < n
      pm : m < n (*1*)
      i := Sub m pm : 'I_n
      ============================
      True
```

The type of pm can be cleaned up by its annotation (*1*) by just simplifying it. The annotations are there for technical reasons only.

When intro patterns for abstract constants are used in conjunction with have and an explicit term, they must be used as follows:

Example

```
Lemma test n m (H : m + 1 < n) : True.
    1 subgoal

      n, m : nat
      H : m + 1 < n
      ============================
      True

have [:pm] @i : 'I_n := Sub m pm.
    2 subgoals

      n, m : nat
      H : m + 1 < n
      ============================
      S m <= n

    subgoal 2 is:
     True

by omega.
    1 subgoal

      n, m : nat
      H : m + 1 < n
      pm : S m <= n (*1*)
      i := Sub m pm : 'I_n : 'I_n
      ============================
      True
```

In this case the abstract constant pm is assigned by using it in the term that follows := and its corresponding goal is left to be solved. Goals corresponding to intro patterns for abstract constants are opened in the order in which the abstract constants are declared (not in the “order” in which they are used in the term).

Note that abstract constants do respect scopes. Hence, if a variable is declared after their introduction, it has to be properly generalized (i.e. explicitly passed to the abstract constant when one makes use of it).

Example

```
Lemma test n m (H : m + 1 < n) : True.
    1 subgoal

      n, m : nat
      H : m + 1 < n
      ============================
      True

have [:pm] @i k : 'I_(n+k) by apply: (Sub m); abstract: pm k; omega.
    1 subgoal

      n, m : nat
      H : m + 1 < n
      pm : (forall k : nat, m < n + k) (*1*)
      i := fun k : nat => Sub m (pm k) : forall k : nat, 'I_(n + k)
      ============================
      True
```

Last, notice that the use of intro patterns for abstract constants is orthogonal to the transparent flag @ for have.

The have tactic and typeclass resolution

Since SSReflect 1.5 the have tactic behaves as follows with respect to typeclass inference.

```
have foo : ty.
    2 subgoals

      ============================
      ty

    subgoal 2 is:
     True
```

Full inference for ty. The first subgoal demands a proof of such instantiated statement.

```
have foo : ty := .
```

No inference for ty. Unresolved instances are quantified in ty. The first subgoal demands a proof of such quantified statement. Note that no proof term follows :=, hence two subgoals are generated.

```
have foo : ty := t.
    1 subgoal

      foo : ty
      ============================
      True
```

No inference for ty and t.

```
have foo := t.
    1 subgoal

      foo : ty
      ============================
      True
```

No inference for `t`. Unresolved instances are quantified in the (inferred) type of `t` and abstracted in `t`.

Flag: `SsrHave NoTCResolution`
This flag restores the behavior of SSReflect 1.4 and below (never resolve typeclasses).

Variants: the suff and wlog tactics

As it is often the case in mathematical textbooks, forward reasoning may be used in slightly different variants. One of these variants is to show that the intermediate step L easily implies the initial goal G. By easily we mean here that the proof of L G is shorter than the one of L itself. This kind of reasoning step usually starts with: "It suffices to show that …".

This is such a frequent way of reasoning that SSReflect has a variant of the `have` tactic called `suffices` (whose abridged name is `suff`). The `have` and `suff` tactics are equivalent and have the same syntax but:

- the order of the generated subgoals is inverted
- the optional clear item is still performed in the *second* branch. This means that the tactic:

  ```
  suff {H} H : forall x : nat, x >= 0.
  ```

  fails if the context of the current goal indeed contains an assumption named `H`.

The rationale of this clearing policy is to make possible "trivial" refinements of an assumption, without changing its name in the main branch of the reasoning.

The `have` modifier can follow the `suff` tactic.

Example

```
Lemma test : G.
    1 subgoal

      ============================
      G

suff have H : P.
    2 subgoals

      H : P
      ============================
      G

    subgoal 2 is:
     (P -> G) -> G
```

Note that, in contrast with `have suff`, the name H has been introduced in the first goal.

Another useful construct is reduction, showing that a particular case is in fact general enough to prove a general property. This kind of reasoning step usually starts with: "Without loss of generality, we can suppose that …". Formally, this corresponds to the proof of a goal `G` by introducing a cut `wlog_statement -> G`. Hence the user shall provide a proof for both `(wlog_statement -> G) -> G` and `wlog_statement -> G`. However, such cuts are usually rather painful to perform by hand, because the statement `wlog_statement` is tedious to write by hand, and sometimes even to read.

SSReflect implements this kind of reasoning step through the *without loss* tactic, whose short name is *wlog*. It offers support to describe the shape of the cut statements, by providing the simplifying hypothesis and by pointing at the elements of the initial goals which should be generalized. The general syntax of without loss is:

`wlog` `suff`? *clear_switch*? *i_item*? `:` *ident** `/` *term*

`without loss` `suff`? *clear_switch*? *i_item*? `:` *ident** `/` *term*

where each *ident* is a constant in the context of the goal. Open syntax is supported for *term*.

In its defective form:

Variant: `wlog: /` *term*
Variant: `without loss: /` *term*

on a goal G, it creates two subgoals: a first one to prove the formula (term -> G) -> G and a second one to prove the formula term -> G.

If the optional list of *ident* is present on the left side of `/`, these constants are generalized in the premise (term -> G) of the first subgoal. By default bodies of local definitions are erased. This behavior can be inhibited by prefixing the name of the local definition with the `@` character.

In the second subgoal, the tactic:

```
move=> clear_switch i_item.
```

is performed if at least one of these optional switches is present in the *wlog* tactic.

The *wlog* tactic is specially useful when a symmetry argument simplifies a proof. Here is an example showing the beginning of the proof that quotient and reminder of natural number euclidean division are unique.

Example

```
Lemma quo_rem_unicity d q1 q2 r1 r2 :
  q1*d + r1 = q2*d + r2 -> r1 < d -> r2 < d -> (q1, r1) = (q2, r2).
    1 subgoal

      d, q1, q2, r1, r2 : nat
      ============================
      q1 * d + r1 = q2 * d + r2 -> r1 < d -> r2 < d -> (q1, r1) = (q2, r2)

wlog: q1 q2 r1 r2 / q1 <= q2.
    2 subgoals

      d, q1, q2, r1, r2 : nat
      ============================
      (forall q3 q4 r3 r4 : nat,
       q3 <= q4 ->
       q3 * d + r3 = q4 * d + r4 -> r3 < d -> r4 < d -> (q3, r3) = (q4, r4)) ->
      q1 * d + r1 = q2 * d + r2 -> r1 < d -> r2 < d -> (q1, r1) = (q2, r2)
```

(continues on next page)

(continued from previous page)

```
    subgoal 2 is:
     q1 <= q2 ->
     q1 * d + r1 = q2 * d + r2 -> r1 < d -> r2 < d -> (q1, r1) = (q2, r2)

by case (le_gt_dec q1 q2)=> H; last symmetry; eauto with arith.
    1 subgoal

      d, q1, q2, r1, r2 : nat
      ============================
      q1 <= q2 ->
      q1 * d + r1 = q2 * d + r2 -> r1 < d -> r2 < d -> (q1, r1) = (q2, r2)
```

The `wlog suff` variant is simpler, since it cuts `wlog_statement` instead of `wlog_statement -> G`. It thus opens the goals `wlog_statement -> G` and `wlog_statement`.

In its simplest form the `generally have : …` tactic is equivalent to `wlog suff : …` followed by last first. When the `have` tactic is used with the `generally` (or `gen`) modifier it accepts an extra identifier followed by a comma before the usual intro pattern. The identifier will name the new hypothesis in its more general form, while the intro pattern will be used to process its instance.

Example

```
Lemma simple n (ngt0 : 0 < n ) : P n.
    1 subgoal

      n : nat
      ngt0 : 0 < n
      ============================
      P n

gen have ltnV, /andP[nge0 neq0] : n ngt0 / (0 <= n) && (n != 0); last first.
    2 subgoals

      n : nat
      ngt0 : 0 < n
      ltnV : forall n : nat, 0 < n -> (0 <= n) && (n != 0)
      nge0 : 0 <= n
      neq0 : n != 0
      ============================
      P n

    subgoal 2 is:
     (0 <= n) && (n != 0)
```

Advanced generalization

The complete syntax for the items on the left hand side of the `/` separator is the following one:

Variant: `wlog … :` [*clear_switch* | [@]? *ident* | ([@]? *ident* := *c_pattern*)]? / *term*

Clear operations are intertwined with generalization operations. This helps in particular avoiding dependency issues while generalizing some facts.

If an *ident* is prefixed with the `@` mark, then a let-in redex is created, which keeps track if its body (if any). The syntax `(`*ident* `:=` *c_pattern*`)` allows to generalize an arbitrary term using a given name. Note that its simplest form `(x := y)` is just a renaming of `y` into `x`. In particular, this can be useful in order to simulate the generalization of a section variable, otherwise not allowed. Indeed renaming does not require the original variable to be cleared.

The syntax `(@x := y)` generates a let-in abstraction but with the following caveat: `x` will not bind `y`, but its body, whenever `y` can be unfolded. This covers the case of both local and global definitions, as illustrated in the following example.

Example

```
Section Test.
Variable x : nat.
    x is declared

Definition addx z := z + x.
    addx is defined

Lemma test : x <= addx x.
    1 subgoal

      x : nat
      ============================
      x <= addx x

wlog H : (y := x) (@twoy := addx x) / twoy = 2 * y.
    2 subgoals

      x : nat
      ============================
      (forall y : nat, let twoy := y + y in twoy = 2 * y -> y <= twoy) ->
      x <= addx x

    subgoal 2 is:
     y <= twoy
```

To avoid unfolding the term captured by the pattern add x one can use the pattern `id (addx x)`, that would produce the following first subgoal

```
wlog H : (y := x) (@twoy := id (addx x)) / twoy = 2 * y.
    2 subgoals

      x : nat
      ============================
      (forall y : nat, let twoy := addx y in twoy = 2 * y -> y <= addx y) ->
      x <= addx x

    subgoal 2 is:
     y <= addx y
```

5.7.6 Rewriting

The generalized use of reflection implies that most of the intermediate results handled are properties of effectively computable functions. The most efficient mean of establishing such results are computation and simplification of expressions involving such functions, i.e., rewriting. SSReflect therefore includes an extended `rewrite` tactic, that unifies and combines most of the rewriting functionalities.

An extended rewrite tactic

The main features of the rewrite tactic are:

- It can perform an entire series of such operations in any subset of the goal and/or context;
- It allows to perform rewriting, simplifications, folding/unfolding of definitions, closing of goals;
- Several rewriting operations can be chained in a single tactic;
- Control over the occurrence at which rewriting is to be performed is significantly enhanced.

The general form of an SSReflect rewrite tactic is:

```
rewrite rstep+
```

The combination of a rewrite tactic with the `in` tactical (see section *Localization*) performs rewriting in both the context and the goal.

A rewrite step *rstep* has the general form:

```
rstep ::= r_prefix? r_item
r_prefix ::= -? mult? occ_switch | clear_switch? [ r_pattern ]?
r_pattern ::= term | in ident in? term | term in | term as ident in term
r_item ::= /? term | s_item
```

An *r_prefix* contains annotations to qualify where and how the rewrite operation should be performed:

- The optional initial `-` indicates the direction of the rewriting of *r_item*: if present the direction is right-to-left and it is left-to-right otherwise.
- The multiplier *mult* (see section *Iteration*) specifies if and how the rewrite operation should be repeated.
- A rewrite operation matches the occurrences of a *rewrite pattern*, and replaces these occurrences by another term, according to the given *r_item*. The optional *redex switch* `[r_pattern]`, which should always be surrounded by brackets, gives explicitly this rewrite pattern. In its simplest form, it is a regular term. If no explicit redex switch is present the rewrite pattern to be matched is inferred from the *r_item*.
- This optional term, or the *r_item*, may be preceded by an *occ_switch* (see section *Selectors*) or a *clear_switch* (see section *Discharge*), these two possibilities being exclusive.

  An occurrence switch selects the occurrences of the rewrite pattern which should be affected by the rewrite operation.

  A clear switch, even an empty one, is performed *after* the *r_item* is actually processed and is complemented with the name of the rewrite rule if an only if it is a simple proof context entry[307]. As a consequence one can write `rewrite {}H` to rewrite with `H` and dispose `H` immediately afterwards. This behavior can be avoided by putting parentheses around the rewrite rule.

An *r_item* can be:

- A *simplification* *r_item*, represented by a *s_item* (see section *Introduction in the context*). Simplification operations are intertwined with the possible other rewrite operations specified by the list of *r_item*.
- A *folding/unfolding* *r_item*. The tactic: `rewrite /term` unfolds the head constant of term in every occurrence of the first matching of term in the goal. In particular, if `my_def` is a (local or global) defined constant, the tactic: `rewrite /my_def.` is analogous to: `unfold my_def`. Conversely: `rewrite -/ my_def.` is equivalent to: `fold my_def`. When an unfold *r_item* is combined with a redex pattern, a conversion operation is performed. A tactic of the form: `rewrite -[term1]/term2.` is equivalent to: `change term1 with term2.` If `term2` is a single constant and `term1` head symbol is not `term2`, then the head symbol of `term1` is repeatedly unfolded until `term2` appears.
- **A *term*, which can be:**
  - A term whose type has the form: `forall (x1 : A1 )…(xn : An ), eq term1 term2` where `eq` is the Leibniz equality or a registered setoid equality.
  - A list of terms `(t1 ,…,tn)`, each `ti` having a type above. The tactic: `rewrite r_prefix (t1 ,…,tn ).` is equivalent to: `do [rewrite r_prefix t1 | … | rewrite r_prefix tn ].`
  - An anonymous rewrite lemma `(_ : term)`, where term has a type as above. tactic: `rewrite (_ : term)` is in fact synonym of: `cutrewrite (term).`.

Example

```
Definition double x := x + x.
    double is defined

Definition ddouble x := double (double x).
    ddouble is defined

Lemma test x : ddouble x = 4 * x.
    1 subgoal

      x : nat
      ============================
      ddouble x = 4 * x

rewrite [ddouble _]/double.
    1 subgoal

      x : nat
      ============================
      double x + double x = 4 * x
```

Warning: The SSReflect terms containing holes are *not* typed as abstractions in this context. Hence the following script fails.

```
Definition f := fun x y => x + y.
    f is defined

Lemma test x y : x + y = f y x.
    1 subgoal

      x, y : nat
      ============================
      x + y = f y x
```

```
rewrite -[f y]/(y + _).
    Toplevel input, characters 0-22:
    > rewrite -[f y]/(y + _).
    > ^^^^^^^^^^^^^^^^^^^^^^
    Error: fold pattern (y + _) does not match redex (f y)
```

but the following script succeeds

```
rewrite -[f y x]/(y + _).
    1 subgoal

      x, y : nat
      ============================
      x + y = y + x
```

Flag: `SsrOldRewriteGoalsOrder`

Controls the order in which generated subgoals (side conditions) are added to the proof context. The flag is off by default, which puts subgoals generated by conditional rules first, followed by the main goal. When it is on, the main goal appears first. If your proofs are organized to complete proving the main goal before side conditions, turning the flag on will save you from having to add *`last first`* tactics that would be needed to keep the main goal as the currently focused goal.

Remarks and examples

Rewrite redex selection

The general strategy of SSReflect is to grasp as many redexes as possible and to let the user select the ones to be rewritten thanks to the improved syntax for the control of rewriting.

This may be a source of incompatibilities between the two rewrite tactics.

In a rewrite tactic of the form:

```
rewrite occ_switch [term1]term2.
```

`term1` is the explicit rewrite redex and `term2` is the rewrite rule. This execution of this tactic unfolds as follows:

- First `term1` and `term2` are $\beta\iota$ normalized. Then `term2` is put in head normal form if the Leibniz equality constructor `eq` is not the head symbol. This may involve ζ reductions.
- Then, the matching algorithm (see section *Abbreviations*) determines the first subterm of the goal matching the rewrite pattern. The rewrite pattern is given by `term1`, if an explicit redex pattern switch is provided, or by the type of `term2` otherwise. However, matching skips over matches that would lead to trivial rewrites. All the occurrences of this subterm in the goal are candidates for rewriting.
- Then only the occurrences coded by *`occ_switch`* (see again section *Abbreviations*) are finally selected for rewriting.
- The left hand side of `term2` is unified with the subterm found by the matching algorithm, and if this succeeds, all the selected occurrences in the goal are replaced by the right hand side of `term2`.
- Finally the goal is $\beta\iota$ normalized.

In the case `term2` is a list of terms, the first top-down (in the goal) left-to-right (in the list) matching rule gets selected.

Chained rewrite steps

The possibility to chain rewrite operations in a single tactic makes scripts more compact and gathers in a single command line a bunch of surgical operations which would be described by a one sentence in a pen and paper proof.

Performing rewrite and simplification operations in a single tactic enhances significantly the concision of scripts. For instance the tactic:

```
rewrite /my_def {2}[f _]/= my_eq //=.
```

unfolds `my_def` in the goal, simplifies the second occurrence of the first subterm matching pattern `[f _]`, rewrites `my_eq`, simplifies the goals and closes trivial goals.

Here are some concrete examples of chained rewrite operations, in the proof of basic results on natural numbers arithmetic.

Example

```
Axiom addn0 : forall m, m + 0 = m.
    addn0 is declared

Axiom addnS : forall m n, m + S n = S (m + n).
    addnS is declared

Axiom addSnnS : forall m n, S m + n = m + S n.
    addSnnS is declared

Lemma addnCA m n p : m + (n + p) = n + (m + p).
    1 subgoal

      m, n, p : nat
      ============================
      m + (n + p) = n + (m + p)

by elim: m p => [ | m Hrec] p; rewrite ?addSnnS -?addnS.
    No more subgoals.

Qed.
Lemma addnC n m : m + n = n + m.
    1 subgoal

      n, m : nat
      ============================
      m + n = n + m

by rewrite -{1}[n]addn0 addnCA addn0.
    No more subgoals.

Qed.
```

Note the use of the `?` switch for parallel rewrite operations in the proof of `addnCA`.

Explicit redex switches are matched first

If an *r_prefix* involves a *redex switch*, the first step is to find a subterm matching this redex pattern, independently from the left hand side of the equality the user wants to rewrite.

Example

```
Lemma test (H : forall t u, t + u = u + t) x y : x + y = y + x.
    1 subgoal

      H : forall t u : nat, t + u = u + t
      x, y : nat
      ============================
      x + y = y + x

rewrite [y + _]H.
    1 subgoal

      H : forall t u : nat, t + u = u + t
      x, y : nat
      ============================
      x + y = x + y
```

Note that if this first pattern matching is not compatible with the *r_item*, the rewrite fails, even if the goal contains a correct redex matching both the redex switch and the left hand side of the equality.

Example

```
Lemma test (H : forall t u, t + u * 0 = t) x y : x + y * 4 + 2 * 0 = x + 2 * 0.
    1 subgoal

      H : forall t u : nat, t + u * 0 = t
      x, y : nat
      ============================
      x + y * 4 + 2 * 0 = x + 2 * 0

Fail rewrite [x + _]H.
    The command has indeed failed with message:
    pattern (x + y * 4) does not match LHS of H
```

Indeed the left hand side of H does not match the redex identified by the pattern x + y * 4.

Occurrence switches and redex switches

Example

```
Lemma test x y : x + y + 0 = x + y + y + 0 + 0 + (x + y + 0).
    1 subgoal

      x, y : nat
      ============================
```

(continues on next page)

(continued from previous page)

```
      x + y + 0 = x + y + y + 0 + 0 + (x + y + 0)

rewrite {2}[_ + y + 0](_: forall z, z + 0 = z).
    2 subgoals

      x, y : nat
      ============================
      forall z : nat, z + 0 = z

    subgoal 2 is:
     x + y + 0 = x + y + y + 0 + 0 + (x + y)
```

The second subgoal is generated by the use of an anonymous lemma in the rewrite tactic. The effect of the tactic on the initial goal is to rewrite this lemma at the second occurrence of the first matching `x + y + 0` of the explicit rewrite redex `_ + y + 0`.

Occurrence selection and repetition

Occurrence selection has priority over repetition switches. This means the repetition of a rewrite tactic specified by a multiplier will perform matching each time an elementary rewrite operation is performed. Repeated rewrite tactics apply to every subgoal generated by the previous tactic, including the previous instances of the repetition.

Example

```
Lemma test x y (z : nat) : x + 1 = x + y + 1.
    1 subgoal

      x, y, z : nat
      ============================
      x + 1 = x + y + 1

rewrite 2!(_ : _ + 1 = z).
    4 subgoals

      x, y, z : nat
      ============================
      x + 1 = z

    subgoal 2 is:
     z = z
    subgoal 3 is:
     x + y + 1 = z
    subgoal 4 is:
     z = z
```

This last tactic generates *three* subgoals because the second rewrite operation specified with the `2!` multiplier applies to the two subgoals generated by the first rewrite.

Multi-rule rewriting

The rewrite tactic can be provided a *tuple* of rewrite rules, or more generally a tree of such rules, since this tuple can feature arbitrary inner parentheses. We call *multirule* such a generalized rewrite rule. This feature is of special interest when it is combined with multiplier switches, which makes the rewrite tactic iterate the rewrite operations prescribed by the rules on the current goal.

Example

```
Variables (a b c : nat).
    a is declared
    b is declared
    c is declared

Hypothesis eqab : a = b.
    eqab is declared

Hypothesis eqac : a = c.
    eqac is declared

Lemma test : a = a.
    1 subgoal

      a, b, c : nat
      eqab : a = b
      eqac : a = c
      ============================
      a = a

rewrite (eqab, eqac).
    1 subgoal

      a, b, c : nat
      eqab : a = b
      eqac : a = c
      ============================
      b = b
```

Indeed rule `eqab` is the first to apply among the ones gathered in the tuple passed to the rewrite tactic. This multirule `(eqab, eqac)` is actually a Coq term and we can name it with a definition:

```
Definition multi1 := (eqab, eqac).
    multi1 is defined
```

In this case, the tactic `rewrite multi1` is a synonym for `rewrite (eqab, eqac)`.

More precisely, a multirule rewrites the first subterm to which one of the rules applies in a left-to-right traversal of the goal, with the first rule from the multirule tree in left-to-right order. Matching is performed according to the algorithm described in Section *Abbreviations*, but literal matches have priority.

Example

```
Definition d := a.
    d is defined
```

(continues on next page)

(continued from previous page)

```
Hypotheses eqd0 : d = 0.
    eqd0 is declared

Definition multi2 := (eqab, eqd0).
    multi2 is defined

Lemma test : d = b.
    1 subgoal

      a, b, c : nat
      eqab : a = b
      eqac : a = c
      eqd0 : d = 0
      ============================
      d = b

rewrite multi2.
    1 subgoal

      a, b, c : nat
      eqab : a = b
      eqac : a = c
      eqd0 : d = 0
      ============================
      0 = b
```

Indeed rule eqd0 applies without unfolding the definition of d.

For repeated rewrites the selection process is repeated anew.

Example

```
Hypothesis eq_adda_b : forall x, x + a = b.
    eq_adda_b is declared

Hypothesis eq_adda_c : forall x, x + a = c.
    eq_adda_c is declared

Hypothesis eqb0 : b = 0.
    eqb0 is declared

Definition multi3 := (eq_adda_b, eq_adda_c, eqb0).
    multi3 is defined

Lemma test : 1 + a = 12 + a.
    1 subgoal

      a, b, c : nat
      eqab : a = b
      eqac : a = c
      eqd0 : d = 0
      eq_adda_b : forall x : nat, x + a = b
      eq_adda_c : forall x : nat, x + a = c
      eqb0 : b = 0
      ============================
```

(continues on next page)

(continued from previous page)

```
      1 + a = 12 + a

rewrite 2!multi3.
    1 subgoal

      a, b, c : nat
      eqab : a = b
      eqac : a = c
      eqd0 : d = 0
      eq_adda_b : forall x : nat, x + a = b
      eq_adda_c : forall x : nat, x + a = c
      eqb0 : b = 0
      ============================
      0 = 12 + a
```

It uses `eq_adda_b` then `eqb0` on the left-hand side only. Without the bound `2` one would obtain `0 = 0`.

The grouping of rules inside a multirule does not affect the selection strategy but can make it easier to include one rule set in another or to (universally) quantify over the parameters of a subset of rules (as there is special code that will omit unnecessary quantifiers for rules that can be syntactically extracted). It is also possible to reverse the direction of a rule subset, using a special dedicated syntax: the tactic rewrite `(=~ multi1)` is equivalent to `rewrite multi1_rev`.

Example

```
Hypothesis eqba : b = a.
    eqba is declared

Hypothesis eqca : c = a.
    eqca is declared

Definition multi1_rev := (eqba, eqca).
    multi1_rev is defined
```

except that the constants `eqba`, `eqab`, `mult1_rev` have not been created.

Rewriting with multirules is useful to implement simplification or transformation procedures, to be applied on terms of small to medium size. For instance the library `ssrnat` (Mathematical Components library) provides two implementations for arithmetic operations on natural numbers: an elementary one and a tail recursive version, less inefficient but also less convenient for reasoning purposes. The library also provides one lemma per such operation, stating that both versions return the same values when applied to the same arguments:

```
Lemma addE : add =2 addn.
Lemma doubleE : double =1 doublen.
Lemma add_mulE n m s : add_mul n m s = addn (muln n m) s.
Lemma mulE : mul =2 muln.
Lemma mul_expE m n p : mul_exp m n p = muln (expn m n) p.
Lemma expE : exp =2 expn.
Lemma oddE : odd =1 oddn.
```

The operation on the left hand side of each lemma is the efficient version, and the corresponding naive implementation is on the right hand side. In order to reason conveniently on expressions involving the efficient operations, we gather all these rules in the definition `trecE`:

```
Definition trecE := (addE, (doubleE, oddE), (mulE, add_mulE, (expE, mul_expE))).
```

The tactic: `rewrite !trecE.` restores the naive versions of each operation in a goal involving the efficient ones, e.g. for the purpose of a correctness proof.

Wildcards vs abstractions

The rewrite tactic supports *r_item*s containing holes. For example, in the tactic `rewrite (_ : _ * 0 = 0).` the term `_ * 0 = 0` is interpreted as `forall n : nat, n * 0 = 0.` Anyway this tactic is *not* equivalent to `rewrite (_ : forall x, x * 0 = 0)..`

Example

```
Lemma test y z : y * 0 + y * (z * 0) = 0.
    1 subgoal

      y, z : nat
      ============================
      y * 0 + y * (z * 0) = 0

rewrite (_ : _ * 0 = 0).
    2 subgoals

      y, z : nat
      ============================
      y * 0 = 0

    subgoal 2 is:
     0 + y * (z * 0) = 0
```

while the other tactic results in

```
rewrite (_ : forall x, x * 0 = 0).
    2 subgoals

      y, z : nat
      ============================
      forall x : nat, x * 0 = 0

    subgoal 2 is:
     0 + y * (z * 0) = 0
```

The first tactic requires you to prove the instance of the (missing) lemma that was used, while the latter requires you prove the quantified form.

When SSReflect rewrite fails on standard Coq licit rewrite

In a few cases, the SSReflect rewrite tactic fails rewriting some redexes which standard Coq successfully rewrites. There are two main cases:

- SSReflect never accepts to rewrite indeterminate patterns like:

```
Lemma foo (x : unit) : x = tt.
```

SSReflect will however accept the $\eta\zeta$ expansion of this rule:

```
Lemma fubar (x : unit) : (let u := x in u) = tt.
```

- The standard rewrite tactic provided by Coq uses a different algorithm to find instances of the rewrite rule.

Example

```
Variable g : nat -> nat.
    g is declared

Definition f := g.
    f is defined

Axiom H : forall x, g x = 0.
    H is declared

Lemma test : f 3 + f 3 = f 6.
    1 subgoal

      g : nat -> nat
      ============================
      f 3 + f 3 = f 6

(* we call the standard rewrite tactic here *)
rewrite -> H.
    1 subgoal

      g : nat -> nat
      ============================
      0 + 0 = f 6
```

This rewriting is not possible in SSReflect because there is no occurrence of the head symbol `f` of the rewrite rule in the goal.

```
rewrite H.
    Toplevel input, characters 0-9:
    > rewrite H.
    > ^^^^^^^^^
    Error: The LHS of H
        (g _)
    does not match any subterm of the goal
```

Rewriting with `H` first requires unfolding the occurrences of `f` where the substitution is to be performed (here there is a single such occurrence), using tactic `rewrite /f` (for a global replacement of f by g) or `rewrite pattern/f`, for a finer selection.

```
rewrite /f H.
    1 subgoal

      g : nat -> nat
      ============================
      0 + 0 = g 6
```

alternatively one can override the pattern inferred from H

```
rewrite [f _]H.
    1 subgoal

      g : nat -> nat
      ============================
      0 + 0 = f 6
```

Existential metavariables and rewriting

The rewrite tactic will not instantiate existing existential metavariables when matching a redex pattern.

If a rewrite rule generates a goal with new existential metavariables in the Prop sort, these will be generalized as for apply (see *The apply tactic*) and corresponding new goals will be generated.

Example

```
Axiom leq : nat -> nat -> bool.
    leq is declared

Notation "m <= n" := (leq m n) : nat_scope.
Notation "m < n"  := (S m <= n) : nat_scope.
Inductive Ord n := Sub x of x < n.
    Ord is defined
    Ord_rect is defined
    Ord_ind is defined
    Ord_rec is defined
    Ord_sind is defined

Notation "'I_ n" := (Ord n) (at level 8, n at level 2, format "''I_' n").
Arguments Sub {_} _ _.
Definition val n (i : 'I_n) := let: Sub a _ := i in a.
    val is defined

Definition insub n x :=
  if @idP (x < n) is ReflectT _ Px then Some (Sub x Px) else None.
    insub is defined

Axiom insubT : forall n x Px, insub n x = Some (Sub x Px).
    insubT is declared

Lemma test (x : 'I_2) y : Some x = insub 2 y.
    1 subgoal

      x : 'I_2
      y : nat
      ============================
      Some x = insub 2 y

rewrite insubT.
    2 subgoals

      x : 'I_2
      y : nat
```

(continues on next page)

(continued from previous page)

```
      ============================
      forall Hyp0 : y < 2, Some x = Some (Sub y Hyp0)

    subgoal 2 is:
     y < 2
```

Since the argument corresponding to Px is not supplied by the user, the resulting goal should be `Some x = Some (Sub y ?Goal)`. Instead, SSReflect `rewrite` tactic hides the existential variable.

As in *The apply tactic*, the `ssrautoprop` tactic is used to try to solve the existential variable.

```
Lemma test (x : 'I_2) y (H : y < 2) : Some x = insub 2 y.
    1 subgoal

      x : 'I_2
      y : nat
      H : y < 2
      ============================
      Some x = insub 2 y

rewrite insubT.
    1 subgoal

      x : 'I_2
      y : nat
      H : y < 2
      ============================
      Some x = Some (Sub y H)
```

As a temporary limitation, this behavior is available only if the rewriting rule is stated using Leibniz equality (as opposed to setoid relations). It will be extended to other rewriting relations in the future.

Rewriting under binders

Goals involving objects defined with higher-order functions often require ”rewriting under binders”. While setoid rewriting is a possible approach in this case, it is common to use regular rewriting along with dedicated extensionality lemmas. This may cause some practical issues during the development of the corresponding scripts, notably as we might be forced to provide the rewrite tactic with complete terms, as shown by the simple example below.

Example

```
Axiom subnn : forall n : nat, n - n = 0.
Parameter map : (nat -> nat) -> list nat -> list nat.
Parameter sumlist : list nat -> nat.
Axiom eq_map :
  forall F1 F2 : nat -> nat,
  (forall n : nat, F1 n = F2 n) ->
  forall l : list nat, map F1 l = map F2 l.


Lemma example_map l : sumlist (map (fun m => m - m) l) = 0.
    1 subgoal
```

(continues on next page)

(continued from previous page)

```
      l : list nat
      ============================
      sumlist (map (fun m : nat => m - m) l) = 0
```

In this context, one cannot directly use eq_map:

```
rewrite eq_map.
    Toplevel input, characters 0-14:
    > rewrite eq_map.
    > ^^^^^^^^^^^^^^
    Error: Unable to find an instance for the variable F2.
    Rule's type:
    (forall F1 F2 : nat -> nat,
     (forall n : nat, F1 n = F2 n) -> forall l : list nat, map F1 l = map F2 l)
```

as we need to explicitly provide the non-inferable argument F2, which corresponds here to the term we want to obtain *after* the rewriting step. In order to perform the rewrite step one has to provide the term by hand as follows:

```
rewrite (@eq_map _ (fun _ : nat => 0)).
    2 subgoals

      l : list nat
      ============================
      forall n : nat, n - n = 0

    subgoal 2 is:
     sumlist (map (fun _ : nat => 0) l) = 0

by move=> m; rewrite subnn.
    1 subgoal

      l : list nat
      ============================
      sumlist (map (fun _ : nat => 0) l) = 0
```

The *under* tactic lets one perform the same operation in a more convenient way:

```
Lemma example_map l : sumlist (map (fun m => m - m) l) = 0.
    1 subgoal

      l : list nat
      ============================
      sumlist (map (fun m : nat => m - m) l) = 0

under eq_map => m do rewrite subnn.
    1 subgoal

      l : list nat
      ============================
      sumlist (map (fun _ : nat => 0) l) = 0
```

The under tactic

The convenience *under* tactic supports the following syntax:

under *r_prefix*? *term* => *i_item*+ ? do *tactic* | [*tactic**|] ?

Operate under the context proved to be extensional by lemma *term*.

Error: Incorrect number of tactics (expected N tactics, was given M).
This error can occur when using the version with a do clause.

The multiplier part of *r_prefix* is not supported.

We distinguish two modes, *interactive mode* without a do clause, and *one-liner mode* with a do clause, which are explained in more detail below.

Interactive mode

Let us redo the running example in interactive mode.

Example

```
Lemma example_map l : sumlist (map (fun m => m - m) l) = 0.
    1 subgoal

      l : list nat
      ============================
      sumlist (map (fun m : nat => m - m) l) = 0

under eq_map => m.
    2 focused subgoals
    (shelved: 2)

      l : list nat
      m : nat
      ============================
      'Under[ m - m ]

    subgoal 2 is:
     sumlist (map ?Goal l) = 0

rewrite subnn.
    2 focused subgoals
    (shelved: 2)

      l : list nat
      m : nat
      ============================
      'Under[ 0 ]

    subgoal 2 is:
     sumlist (map ?Goal l) = 0

over.
    1 focused subgoal
    (shelved: 1)

      l : list nat
      ============================
      sumlist (map (fun _ : nat => 0) l) = 0
```

The execution of the Ltac expression:

`under` *term* `=> [` *i_item₁* `| … |` *i_itemₙ* `].`

involves the following steps:

1. It performs a `rewrite` *term* without failing like in the first example with `rewrite eq_map.`, but creating evars (see *evar*). If `term` is prefixed by a pattern or an occurrence selector, then the modifiers are honoured.
2. As a n-branches intro pattern is provided *under* checks that n+1 subgoals have been created. The last one is the main subgoal, while the other ones correspond to premises of the rewrite rule (such as `forall n, F1 n = F2 n` for `eq_map`).
3. If so *under* puts these n goals in head normal form (using the defective form of the tactic *move*), then executes the corresponding intro pattern *i_pattern$_i$* in each goal.
4. Then *under* checks that the first n subgoals are (quantified) Leibniz equalities, double implications or registered relations (w.r.t. Class `RewriteRelation`) between a term and an evar, e.g. `m - m = ?F2 m` in the running example. (This support for setoid-like relations is enabled as soon as we do both `Require Import ssreflect.` and `Require Setoid.`)
5. If so *under* protects these n goals against an accidental instantiation of the evar. These protected goals are displayed using the `'Under[ … ]` notation (e.g. `'Under[ m - m ]` in the running example).
6. The expression inside the `'Under[ … ]` notation can be proved equivalent to the desired expression by using a regular *rewrite* tactic.
7. Interactive editing of the first n goals has to be signalled by using the *over* tactic or rewrite rule (see below), which requires that the underlying relation is reflexive. (The running example deals with Leibniz equality, but `PreOrder` relations are also supported, for example.)
8. Finally, a post-processing step is performed in the main goal to keep the name(s) for the bound variables chosen by the user in the intro pattern for the first branch.

The over tactic

Two equivalent facilities (a terminator and a lemma) are provided to close intermediate subgoals generated by *under* (i.e. goals displayed as `'Under[ … ]`):

`over`

This terminator tactic allows one to close goals of the form `'Under[ … ]`.

Variant: `by rewrite over`

This is a variant of *over* in order to close `'Under[ … ]` goals, relying on the `over` rewrite rule.

Note that a rewrite rule `UnderE` is available as well, if one wants to "unprotect" the evar, without closing the goal automatically (e.g., to instantiate it manually with another rule than reflexivity).

One-liner mode

The Ltac expression:

`under` *term* `=> [` *i_item₁* `| … |` *i_itemₙ* `] do [` *tactic₁* `| … |` *tacticₙ* `].`

can be seen as a shorter form for the following expression:

(under *term*) => [*i_item*$_1$ | … | *i_item*$_n$ |]; [*tactic*$_1$; over | … | *tactic*$_n$; over | cbv beta iota].

Notes:

- The beta-iota reduction here is useful to get rid of the beta redexes that could be introduced after the substitution of the evars by the *under* tactic.
- Note that the provided tactics can as well involve other *under* tactics. See below for a typical example involving the bigop theory from the Mathematical Components library.
- If there is only one tactic, the brackets can be omitted, e.g.: under *term* => i do *tactic*. and that shorter form should be preferred.
- If the do clause is provided and the intro pattern is omitted, then the default *i_item* * is applied to each branch. E.g., the Ltac expression: under *term* do [*tactic*$_1$ | … | *tactic*$_n$] is equivalent to: under *term* => [* | … | *] do [*tactic*$_1$ | … | *tactic*$_n$] (and it can be noted here that the *under* tactic performs a move. before processing the intro patterns => [* | … | *]).

Example

```
Parameter addnC : forall m n : nat, m + n = n + m.
Parameter muln1 : forall n : nat, n * 1 = n.


Check eq_bigr.
    eq_bigr
         : forall (n m : nat) (P : nat -> bool) (F1 F2 : nat -> nat),
           (forall x : nat, P x -> F1 x = F2 x) ->
           \sum_(n <= i < m | P i) F1 i = \sum_(n <= i < m | P i) F2 i


Check eq_big.
    eq_big
         : forall (n m : nat) (P1 P2 : nat -> bool) (F1 F2 : nat -> nat),
           (forall x : nat, P1 x = P2 x) ->
           (forall i : nat, P1 i -> F1 i = F2 i) ->
           \sum_(n <= i < m | P1 i) F1 i = \sum_(n <= i < m | P2 i) F2 i


Lemma test_big_nested (m n : nat) :
  \sum_(0 <= a < m | prime a) \sum_(0 <= j < n | odd (j * 1)) (a + j) =
  \sum_(0 <= i < m | prime i) \sum_(0 <= j < n | odd j) (j + i).
    1 subgoal

      m, n : nat
      ============================
      \sum_(0 <= a < m | prime a) \sum_(0 <= j < n | odd (j * 1)) (a + j) =
      \sum_(0 <= i < m | prime i) \sum_(0 <= j < n | odd j) (j + i)


under eq_bigr => i prime_i do
  under eq_big => [ j | j odd_j ] do
    [ rewrite (muln1 j) | rewrite (addnC i j) ].
    1 subgoal

      m, n : nat
      ============================
      \sum_(0 <= i < m | prime i) \sum_(0 <= j < n | odd j) (j + i) =
      \sum_(0 <= i < m | prime i) \sum_(0 <= j < n | odd j) (j + i)
```

Remark how the final goal uses the name i (the name given in the intro pattern) rather than a in the binder of the first summation.

Locking, unlocking

As program proofs tend to generate large goals, it is important to be able to control the partial evaluation performed by the simplification operations that are performed by the tactics. These evaluations can for example come from a `/=` simplification switch, or from rewrite steps which may expand large terms while performing conversion. We definitely want to avoid repeating large subterms of the goal in the proof script. We do this by "clamping down" selected function symbols in the goal, which prevents them from being considered in simplification or rewriting steps. This clamping is accomplished by using the occurrence switches (see section *Abbreviations*) together with "term tagging" operations.

SSReflect provides two levels of tagging.

The first one uses auxiliary definitions to introduce a provably equal copy of any term t. However this copy is (on purpose) *not convertible* to t in the Coq system[305]. The job is done by the following construction:

```
Lemma master_key : unit. Proof. exact tt. Qed.
Definition locked A := let: tt := master_key in fun x : A => x.
Lemma lock : forall A x, x = locked x :> A.
```

Note that the definition of *master_key* is explicitly opaque. The equation `t = locked t` given by the `lock` lemma can be used for selective rewriting, blocking on the fly the reduction in the term `t`.

Example

```
Variable A : Type.
    A is declared

Fixpoint has (p : A -> bool) (l : list A) : bool :=
  if l is cons x l then p x || (has p l) else false.
    has is defined
    has is recursively defined (decreasing on 2nd argument)

Lemma test p x y l (H : p x = true) : has p ( x :: y :: l) = true.
    1 subgoal

      A : Type
      p : A -> bool
      x, y : A
      l : list A
      H : p x = true
      ============================
      has p (x :: y :: l) = true

rewrite {2}[cons]lock /= -lock.
    1 subgoal

      A : Type
      p : A -> bool
      x, y : A
      l : list A
      H : p x = true
```

(continues on next page)

[305] This is an implementation feature: there is no such obstruction in the metatheory

(continued from previous page)

```
      ============================
      p x || has p (y :: l) = true
```

It is sometimes desirable to globally prevent a definition from being expanded by simplification; this is done by adding locked in the definition.

Example

```
Definition lid := locked (fun x : nat => x).
    lid is defined

Lemma test : lid 3 = 3.
    1 subgoal

      ============================
      lid 3 = 3

rewrite /=.
    1 subgoal

      ============================
      lid 3 = 3

unlock lid.
    1 subgoal

      ============================
      3 = 3
```

unlock *occ_switch*? *ident*

This tactic unfolds such definitions while removing "locks", i.e. it replaces the occurrence(s) of *ident* coded by the *occ_switch* with the corresponding body.

We found that it was usually preferable to prevent the expansion of some functions by the partial evaluation switch `/=`, unless this allowed the evaluation of a condition. This is possible thanks to another mechanism of term tagging, resting on the following *Notation*:

```
Notation "'nosimpl' t" := (let: tt := tt in t).
```

The term `(nosimpl t)` simplifies to `t` *except* in a definition. More precisely, given:

```
Definition foo := (nosimpl bar).
```

the term `foo` (or `(foo t')`) will *not* be expanded by the *simpl* tactic unless it is in a forcing context (e.g., in `match foo t' with … end`, `foo t'` will be reduced if this allows `match` to be reduced). Note that `nosimpl bar` is simply notation for a term that reduces to `bar`; hence `unfold foo` will replace `foo` by `bar`, and `fold foo` will replace `bar` by `foo`.

Warning: The `nosimpl` trick only works if no reduction is apparent in `t`; in particular, the declaration:

```
Definition foo x := nosimpl (bar x).
```

will usually not work. Anyway, the common practice is to tag only the function, and to use the following definition, which blocks the reduction as expected:

```
Definition foo x := nosimpl bar x.
```

A standard example making this technique shine is the case of arithmetic operations. We define for instance:

```
Definition addn := nosimpl plus.
```

The operation `addn` behaves exactly like `plus`, except that `(addn (S n) m)` will not simplify spontaneously to `(S (addn n m))` (the two terms, however, are convertible). In addition, the unfolding step: `rewrite /addn` will replace `addn` directly with `plus`, so the `nosimpl` form is essentially invisible.

Congruence

Because of the way matching interferes with parameters of type families, the tactic:

```
apply: my_congr_property.
```

will generally fail to perform congruence simplification, even on rather simple cases. We therefore provide a more robust alternative in which the function is supplied:

congr *num*? *term*

This tactic:

- checks that the goal is a Leibniz equality;
- matches both sides of this equality with "term applied to some arguments", inferring the right number of arguments from the goal and the type of term. This may expand some definitions or fixpoints;
- generates the subgoals corresponding to pairwise equalities of the arguments present in the goal.

The goal can be a non dependent product `P -> Q`. In that case, the system asserts the equation `P = Q`, uses it to solve the goal, and calls the `congr` tactic on the remaining goal `P = Q`. This can be useful for instance to perform a transitivity step, like in the following situation.

Example

```
Lemma test (x y z : nat) (H : x = y) : x = z.
    1 subgoal

      x, y, z : nat
      H : x = y
      ============================
      x = z

congr (_ = _) : H.
    1 focused subgoal
    (shelved: 1)

      x, y, z : nat
      ============================
      y = z
```

(continues on next page)

(continued from previous page)

```
Abort.
Lemma test (x y z : nat) : x = y -> x = z.
    1 subgoal

      x, y, z : nat
      ============================
      x = y -> x = z

congr (_ = _).
    1 focused subgoal
    (shelved: 1)

      x, y, z : nat
      ============================
      y = z
```

The optional *num* forces the number of arguments for which the tactic should generate equality proof obligations.

This tactic supports equalities between applications with dependent arguments. Yet dependent arguments should have exactly the same parameters on both sides, and these parameters should appear as first arguments.

Example

```
Definition f n :=
  if n is 0 then plus else mult.
    f is defined

Definition g (n m : nat) := plus.
    g is defined

Lemma test x y : f 0 x y = g 1 1 x y.
    1 subgoal

      x, y : nat
      ============================
      f 0 x y = g 1 1 x y

congr plus.
    No more subgoals.
```

This script shows that the `congr` tactic matches `plus` with `f 0` on the left hand side and `g 1 1` on the right hand side, and solves the goal.

Example

```
Lemma test n m (Hnm : m <= n) : S m + (S n - S m) = S n.
    1 subgoal

      n, m : nat
      Hnm : m <= n
      ============================
```

(continues on next page)

(continued from previous page)

```
      S m + (S n - S m) = S n

congr S; rewrite -/plus.
    1 subgoal

      n, m : nat
      Hnm : m <= n
      ============================
      m + (S n - S m) = n
```

The tactic `rewrite -/plus` folds back the expansion of plus which was necessary for matching both sides of the equality with an application of `S`.

Like most SSReflect arguments, *term* can contain wildcards.

Example

```
Lemma test x y : x + (y * (y + x - x)) = x * 1 + (y + 0) * y.
    1 subgoal

      x, y : nat
      ============================
      x + y * (y + x - x) = x * 1 + (y + 0) * y

congr ( _ + (_ * _)).
    3 focused subgoals
    (shelved: 3)

      x, y : nat
      ============================
      x = x * 1

    subgoal 2 is:
     y = y + 0
    subgoal 3 is:
     y + x - x = y
```

5.7.7 Contextual patterns

The simple form of patterns used so far, terms possibly containing wild cards, often require an additional *occ_switch* to be specified. While this may work pretty fine for small goals, the use of polymorphic functions and dependent types may lead to an invisible duplication of function arguments. These copies usually end up in types hidden by the implicit arguments machinery or by user-defined notations. In these situations computing the right occurrence numbers is very tedious because they must be counted on the goal as printed after setting the *Printing All* flag. Moreover the resulting script is not really informative for the reader, since it refers to occurrence numbers he cannot easily see.

Contextual patterns mitigate these issues allowing to specify occurrences according to the context they occur in.

Syntax

The following table summarizes the full syntax of *c_pattern* and the corresponding subterm(s) identified by the pattern. In the third column we use s.m.r. for "the subterms matching the redex" specified in the second column.

c_pattern	redex	subterms affected
`term`	`term`	all occurrences of `term`
`ident in term`	subterm of `term` selected by `ident`	all the subterms identified by `ident` in all the occurrences of `term`
`term1 in ident in term2`	`term1` in all s.m.r.	in all the subterms identified by `ident` in all the occurrences of `term2`
`term1 as ident in term2`	`term 1`	in all the subterms identified by `ident` in all the occurrences of `term2[term 1 /ident]`

The rewrite tactic supports two more patterns obtained prefixing the first two with in. The intended meaning is that the pattern identifies all subterms of the specified context. The `rewrite` tactic will infer a pattern for the redex looking at the rule used for rewriting.

r_pattern	redex	subterms affected
`in term`	inferred from rule	in all s.m.r. in all occurrences of `term`
`in ident in term`	inferred from rule	in all s.m.r. in all the subterms identified by `ident` in all the occurrences of `term`

The first *c_pattern* is the simplest form matching any context but selecting a specific redex and has been described in the previous sections. We have seen so far that the possibility of selecting a redex using a term with holes is already a powerful means of redex selection. Similarly, any terms provided by the user in the more complex forms of *c_pattern*s presented in the tables above can contain holes.

For a quick glance at what can be expressed with the last *r_pattern* consider the goal `a = b` and the tactic

```
rewrite [in X in _ = X]rule.
```

It rewrites all occurrences of the left hand side of `rule` inside `b` only (`a`, and the hidden type of the equality, are ignored). Note that the variant `rewrite [X in _ = X]rule` would have rewritten `b` exactly (i.e., it would only work if `b` and the left hand side of rule can be unified).

Matching contextual patterns

The *c_pattern* and *r_pattern* involving terms with holes are matched against the goal in order to find a closed instantiation. This matching proceeds as follows:

c_pattern	instantiation order and place for `term_i` and redex
`term`	`term` is matched against the goal, redex is unified with the instantiation of `term`
`ident in term`	`term` is matched against the goal, redex is unified with the subterm of the instantiation of `term` identified by `ident`
`term1 in ident in term2`	`term2` is matched against the goal, `term1` is matched against the subterm of the instantiation of `term1` identified by `ident`, redex is unified with the instantiation of `term1`
`term1 as ident in term2`	`term2[term1/ident]` is matched against the goal, redex is unified with the instantiation of `term1`

In the following patterns, the redex is intended to be inferred from the rewrite rule.

r_pattern	instantiation order and place for `term_i` and redex
`in ident in term`	`term` is matched against the goal, the redex is matched against the subterm of the instantiation of `term` identified by `ident`
`in term`	`term` is matched against the goal, redex is matched against the instantiation of `term`

Examples

Contextual pattern in set and the : tactical

As already mentioned in section *Abbreviations* the `set` tactic takes as an argument a term in open syntax. This term is interpreted as the simplest form of *c_pattern*. To avoid confusion in the grammar, open syntax is supported only for the simplest form of patterns, while parentheses are required around more complex patterns.

Example

```
Lemma test a b : a + b + 1 = b + (a + 1).
    1 subgoal

      a, b : nat
      ============================
      a + b + 1 = b + (a + 1)

set t := (X in _ = X).
    1 subgoal

      a, b : nat
      t := b + (a + 1) : nat
      ============================
      a + b + 1 = t

rewrite {}/t.
    1 subgoal

      a, b : nat
      ============================
      a + b + 1 = b + (a + 1)

set t := (a + _ in X in _ = X).
    1 subgoal

      a, b : nat
      t := a + 1 : nat
      ============================
      a + b + 1 = b + t
```

Since the user may define an infix notation for `in` the result of the former tactic may be ambiguous. The disambiguation rule implemented is to prefer patterns over simple terms, but to interpret a pattern with double parentheses as a simple term. For example, the following tactic would capture any occurrence of the term `a in A`.

```
set t := ((a in A)).
```

Contextual patterns can also be used as arguments of the `:` tactical. For example:

```
elim: n (n in _ = n) (refl_equal n).
```

Contextual patterns in rewrite

Example

```
Notation "n .+1" := (Datatypes.S n) (at level 2, left associativity,
                     format "n .+1") : nat_scope.
Axiom addSn : forall m n, m.+1 + n = (m + n).+1.
    addSn is declared

Axiom addn0 : forall m, m + 0 = m.
    addn0 is declared

Axiom addnC : forall m n, m + n = n + m.
    addnC is declared

Lemma test x y z f : (x.+1 + y) + f (x.+1 + y) (z + (x + y).+1) = 0.
    1 subgoal

      x, y, z : nat
      f : nat -> nat -> nat
      ============================
      x.+1 + y + f (x.+1 + y) (z + (x + y).+1) = 0

rewrite [in f _ _]addSn.
    1 subgoal

      x, y, z : nat
      f : nat -> nat -> nat
      ============================
      x.+1 + y + f (x + y).+1 (z + (x + y).+1) = 0
```

Note: the simplification rule `addSn` is applied only under the `f` symbol. Then we simplify also the first addition and expand `0` into `0 + 0`.

```
rewrite addSn -[X in _ = X]addn0.
    1 subgoal

      x, y, z : nat
      f : nat -> nat -> nat
      ============================
      (x + y).+1 + f (x + y).+1 (z + (x + y).+1) = 0 + 0
```

Note that the right hand side of `addn0` is undetermined, but the rewrite pattern specifies the redex explicitly. The right hand side of `addn0` is unified with the term identified by `X`, here `0`.

The following pattern does not specify a redex, since it identifies an entire region, hence the rewrite rule has to be instantiated explicitly. Thus the tactic:

```
rewrite -{2}[in X in _ = X](addn0 0).
    1 subgoal

      x, y, z : nat
      f : nat -> nat -> nat
      ============================
      (x + y).+1 + f (x + y).+1 (z + (x + y).+1) = 0 + (0 + 0)
```

The following tactic is quite tricky:

```
rewrite [_.+1 in X in f _ X](addnC x.+1).
    1 subgoal

      x, y, z : nat
      f : nat -> nat -> nat
      ============================
      (x + y).+1 + f (x + y).+1 (z + (y + x.+1)) = 0 + (0 + 0)
```

The explicit redex `_.+1` is important since its head constant `S` differs from the head constant inferred from `(addnC x.+1)` (that is `+`). Moreover, the pattern `f _ X` is important to rule out the first occurrence of `(x + y).+1`. Last, only the subterms of `f _ X` identified by `X` are rewritten, thus the first argument of `f` is skipped too. Also note the pattern `_.+1` is interpreted in the context identified by `X`, thus it gets instantiated to `(y + x).+1` and not `(x + y).+1`.

The last rewrite pattern allows to specify exactly the shape of the term identified by X, that is thus unified with the left hand side of the rewrite rule.

```
rewrite [x.+1 + y as X in f X _]addnC.
    1 subgoal

      x, y, z : nat
      f : nat -> nat -> nat
      ============================
      (x + y).+1 + f (y + x.+1) (z + (y + x.+1)) = 0 + (0 + 0)
```

Patterns for recurrent contexts

The user can define shortcuts for recurrent contexts corresponding to the `ident in term` part. The notation scope identified with `%pattern` provides a special notation `(X in t)` the user must adopt in order to define context shortcuts.

The following example is taken from `ssreflect.v` where the `LHS` and `RHS` shortcuts are defined.

```
Notation RHS := (X in _ = X)%pattern.
Notation LHS := (X in X = _)%pattern.
```

Shortcuts defined this way can be freely used in place of the trailing `ident in term` part of any contextual pattern. Some examples follow:

```
set rhs := RHS.
rewrite [in RHS]rule.
case: (a + _ in RHS).
```

5.7.8 Views and reflection

The bookkeeping facilities presented in section *Basic tactics* are crafted to ease simultaneous introductions and generalizations of facts and operations of casing, naming etc. It also a common practice to make a stack operation immediately followed by an *interpretation* of the fact being pushed, that is, to apply a lemma to this fact before passing it to a tactic for decomposition, application and so on.

SSReflect provides a convenient, unified syntax to combine these interpretation operations with the proof stack operations. This *view mechanism* relies on the combination of the `/` view switch with bookkeeping tactics and tacticals.

Interpreting eliminations

The view syntax combined with the `elim` tactic specifies an elimination scheme to be used instead of the default, generated, one. Hence the SSReflect tactic:

```
elim/V.
```

is a synonym for:

```
intro top; elim top using V; clear top.
```

where top is a fresh name and V any second-order lemma.

Since an elimination view supports the two bookkeeping tacticals of discharge and introduction (see section *Basic tactics*), the SSReflect tactic:

```
elim/V: x => y.
```

is a synonym for:

```
elim x using V; clear x; intro y.
```

where `x` is a variable in the context, `y` a fresh name and `V` any second order lemma; SSReflect relaxes the syntactic restrictions of the Coq `elim`. The first pattern following `:` can be a `_` wildcard if the conclusion of the view `V` specifies a pattern for its last argument (e.g., if `V` is a functional induction lemma generated by the `Function` command).

The elimination view mechanism is compatible with the equation name generation (see section *Generation of equations*).

Example

The following script illustrates a toy example of this feature. Let us define a function adding an element at the end of a list:

```
Variable d : Type.
    d is declared

Fixpoint add_last (s : list d) (z : d) {struct s} : list d :=
  if s is cons x s' then cons x (add_last s' z) else z :: nil.
    add_last is defined
    add_last is recursively defined (decreasing on 1st argument)
```

One can define an alternative, reversed, induction principle on inductively defined lists, by proving the following lemma:

```
Axiom last_ind_list : forall P : list d -> Prop,
  P nil -> (forall s (x : d), P s -> P (add_last s x)) ->
    forall s : list d, P s.
    last_ind_list is declared
```

Then the combination of elimination views with equation names result in a concise syntax for reasoning inductively using the user-defined elimination scheme.

```
Lemma test (x : d) (l : list d): l = l.
    1 subgoal

      d : Type
      x : d
      l : list d
      ============================
      l = l

elim/last_ind_list E : l=> [| u v]; last first.
    2 subgoals

      d : Type
      x : d
      u : list d
      v : d
      l : list d
      E : l = add_last u v
      ============================
      u = u -> add_last u v = add_last u v

    subgoal 2 is:
     nil = nil
```

User-provided eliminators (potentially generated with Coq's `Function` command) can be combined with the type family switches described in section *Type families*. Consider an eliminator `foo_ind` of type:

```
foo_ind : forall …, forall x : T, P p1 … pm.
```

and consider the tactic:

```
elim/foo_ind: e1 … / en.
```

The `elim/` tactic distinguishes two cases:

truncated eliminator when `x` does not occur in `P p1 … pm` and the type of `en` unifies with `T` and `en` is not `_`. In that case, `en` is passed to the eliminator as the last argument (`x` in `foo_ind`) and `en-1 … e1` are used as patterns to select in the goal the occurrences that will be bound by the predicate `P`, thus it must be possible to unify the subterm of the goal matched by `en-1` with `pm` , the one matched by `en-2` with `pm-1` and so on.

regular eliminator in all the other cases. Here it must be possible to unify the term matched by `en` with `pm` , the one matched by `en-1` with `pm-1` and so on. Note that standard eliminators have the shape `…forall x, P … x`, thus `en` is the pattern identifying the eliminated term, as expected.

As explained in section *Type families*, the initial prefix of `ei` can be omitted.

Here is an example of a regular, but nontrivial, eliminator.

Example

Here is a toy example illustrating this feature.

```
Function plus (m n : nat) {struct n} : nat :=
  if n is S p then S (plus m p) else m.
    plus is defined
    plus is recursively defined (decreasing on 2nd argument)
    plus_equation is defined
    plus_rect is defined
    plus_ind is defined
    plus_rec is defined
    R_plus_correct is defined
    R_plus_complete is defined

About plus_ind.
    plus_ind :
    forall (m : nat) (P : nat -> nat -> Prop),
    (forall n p : nat, n = S p -> P p (plus m p) -> P (S p) (S (plus m p))) ->
    (forall n _x : nat,
     n = _x -> match _x with
               | 0 => True
               | S _ => False
               end -> P _x m) -> forall n : nat, P n (plus m n)

    plus_ind is not universe polymorphic
    Arguments plus_ind [m]%nat_scope [P]%function_scope (_ _)%function_scope
      _%nat_scope
    plus_ind is transparent
    Expands to: Constant Top.Test.plus_ind

Lemma test x y z : plus (plus x y) z = plus x (plus y z).
    1 subgoal

      x, y, z : nat
      ============================
      plus (plus x y) z = plus x (plus y z)
```

The following tactics are all valid and perform the same elimination on this goal.

```
elim/plus_ind: z / (plus _ z).
elim/plus_ind: {z}(plus _ z).
elim/plus_ind: {z}_.
elim/plus_ind: z / _.
```

```
elim/plus_ind: z / _.
    2 subgoals

      x, y : nat
      ============================
      forall n p : nat,
      n = S p ->
      plus (plus x y) p = plus x (plus y p) ->
      S (plus (plus x y) p) = plus x (plus y (S p))

    subgoal 2 is:
     forall n _x : nat,
```

(continues on next page)

(continued from previous page)

```
    n = _x ->
    match _x with
    | 0 => True
    | S _ => False
    end -> plus x y = plus x (plus y _x)
```

The two latter examples feature a wildcard pattern: in this case, the resulting pattern is inferred from the type of the eliminator. In both these examples, it is (plus _ _), which matches the subterm plus (plus x y) z thus instantiating the last _ with z. Note that the tactic:

```
Fail elim/plus_ind: y / _.
    The command has indeed failed with message:
    The given pattern matches the term y while the inferred pattern z doesn't
```

triggers an error: in the conclusion of the plus_ind eliminator, the first argument of the predicate P should be the same as the second argument of plus, in the second argument of P, but y and z do no unify.

Here is an example of a truncated eliminator:

Example

Consider the goal:

```
Lemma test p n (n_gt0 : 0 < n) (pr_p : prime p) :
  p %| \prod_(i <- prime_decomp n | i \in prime_decomp n) i.1 ^ i.2 ->
    exists2 x : nat * nat, x \in prime_decomp n & p = x.1.
Proof.
elim/big_prop: _ => [| u v IHu IHv | [q e] /=].
```

where the type of the big_prop eliminator is

```
big_prop: forall (R : Type) (Pb : R -> Type)
  (idx : R) (op1 : R -> R -> R), Pb idx ->
  (forall x y : R, Pb x -> Pb y -> Pb (op1 x y)) ->
  forall (I : Type) (r : seq I) (P : pred I) (F : I -> R),
  (forall i : I, P i -> Pb (F i)) ->
    Pb (\big[op1/idx]_(i <- r | P i) F i).
```

Since the pattern for the argument of Pb is not specified, the inferred one is used instead: big[_/_]_(i <- _ | _ i) _ i, and after the introductions, the following goals are generated:

```
subgoal 1 is:
  p %| 1 -> exists2 x : nat * nat, x \in prime_decomp n & p = x.1
subgoal 2 is:
  p %| u * v -> exists2 x : nat * nat, x \in prime_decomp n & p = x.1
subgoal 3 is:
  (q, e) \in prime_decomp n -> p %| q ^ e ->
    exists2 x : nat * nat, x \in prime_decomp n & p = x.1.
```

Note that the pattern matching algorithm instantiated all the variables occurring in the pattern.

Interpreting assumptions

Interpreting an assumption in the context of a proof consists in applying to it a lemma before generalizing, and/or decomposing this assumption. For instance, with the extensive use of boolean reflection (see section *Views and reflection*), it is quite frequent to need to decompose the logical interpretation of (the boolean expression of) a fact, rather than the fact itself. This can be achieved by a combination of `move : _ => _` switches, like in the following example, where `||` is a notation for the boolean disjunction.

Example

```
Variables P Q : bool -> Prop.
    P is declared
    Q is declared

Hypothesis P2Q : forall a b, P (a || b) -> Q a.
    P2Q is declared

Lemma test a : P (a || a) -> True.
    1 subgoal

      P, Q : bool -> Prop
      P2Q : forall a b : bool, P (a || b) -> Q a
      a : bool
      ============================
      P (a || a) -> True

move=> HPa; move: {HPa}(P2Q HPa) => HQa.
    1 subgoal

      P, Q : bool -> Prop
      P2Q : forall a b : bool, P (a || b) -> Q a
      a : bool
      HQa : Q a
      ============================
      True
```

which transforms the hypothesis `HPa : P a` which has been introduced from the initial statement into `HQa : Q a`. This operation is so common that the tactic shell has specific syntax for it. The following scripts:

```
move=> HPa; move/P2Q: HPa => HQa.
    1 subgoal

      P, Q : bool -> Prop
      P2Q : forall a b : bool, P (a || b) -> Q a
      a : bool
      HQa : Q a
      ============================
      True
```

or more directly:

```
move/P2Q=> HQa.
    1 subgoal

      P, Q : bool -> Prop
      P2Q : forall a b : bool, P (a || b) -> Q a
      a : bool
```

(continues on next page)

(continued from previous page)

```
      HQa : Q a
      ============================
      True
```

are equivalent to the former one. The former script shows how to interpret a fact (already in the context), thanks to the discharge tactical (see section *Discharge*) and the latter, how to interpret the top assumption of a goal. Note that the number of wildcards to be inserted to find the correct application of the view lemma to the hypothesis has been automatically inferred.

The view mechanism is compatible with the case tactic and with the equation name generation mechanism (see section *Generation of equations*):

Example

```
Variables P Q: bool -> Prop.
    P is declared
    Q is declared

Hypothesis Q2P : forall a b, Q (a || b) -> P a \/ P b.
    Q2P is declared

Lemma test a b : Q (a || b) -> True.
    1 subgoal

      P, Q : bool -> Prop
      Q2P : forall a b : bool, Q (a || b) -> P a \/ P b
      a, b : bool
      ============================
      Q (a || b) -> True

case/Q2P=> [HPa | HPb].
    2 subgoals

      P, Q : bool -> Prop
      Q2P : forall a b : bool, Q (a || b) -> P a \/ P b
      a, b : bool
      HPa : P a
      ============================
      True

    subgoal 2 is:
     True
```

This view tactic performs:

```
move=> HQ; case: {HQ}(Q2P HQ) => [HPa | HPb].
```

The term on the right of the / view switch is called a *view lemma*. Any SSReflect term coercing to a product type can be used as a view lemma.

The examples we have given so far explicitly provide the direction of the translation to be performed. In fact, view lemmas need not to be oriented. The view mechanism is able to detect which application is relevant for the current goal.

Example

```
Variables P Q: bool -> Prop.
    P is declared
    Q is declared

Hypothesis PQequiv : forall a b, P (a || b) <-> Q a.
    PQequiv is declared

Lemma test a b : P (a || b) -> True.
    1 subgoal

      P, Q : bool -> Prop
      PQequiv : forall a b : bool, P (a || b) <-> Q a
      a, b : bool
      ============================
      P (a || b) -> True

move/PQequiv=> HQab.
    1 subgoal

      P, Q : bool -> Prop
      PQequiv : forall a b : bool, P (a || b) <-> Q a
      a, b : bool
      HQab : Q a
      ============================
      True
```

has the same behavior as the first example above.

The view mechanism can insert automatically a *view hint* to transform the double implication into the expected simple implication. The last script is in fact equivalent to:

```
Lemma test a b : P (a || b) -> True.
move/(iffLR (PQequiv _ _)).
```

where:

```
Lemma iffLR P Q : (P <-> Q) -> P -> Q.
```

Specializing assumptions

The special case when the *head symbol* of the view lemma is a wildcard is used to interpret an assumption by *specializing* it. The view mechanism hence offers the possibility to apply a higher-order assumption to some given arguments.

Example

```
Lemma test z : (forall x y, x + y = z -> z = x) -> z = 0.
    1 subgoal

      z : nat
      ============================
```

(continues on next page)

(continued from previous page)

```
      (forall x y : nat, x + y = z -> z = x) -> z = 0

move/(_ 0 z).
    1 subgoal

      z : nat
      ============================
      (0 + z = z -> z = 0) -> z = 0
```

Interpreting goals

In a similar way, it is also often convenient to changing a goal by turning it into an equivalent proposition. The view mechanism of SSReflect has a special syntax `apply/` for combining in a single tactic simultaneous goal interpretation operations and bookkeeping steps.

Example

The following example use the `~~` prenex notation for boolean negation:

```
Variables P Q: bool -> Prop.
    P is declared
    Q is declared

Hypothesis PQequiv : forall a b, P (a || b) <-> Q a.
    PQequiv is declared

Lemma test a : P ((~~ a) || a).
    1 subgoal

      P, Q : bool -> Prop
      PQequiv : forall a b : bool, P (a || b) <-> Q a
      a : bool
      ============================
      P (~~ a || a)

apply/PQequiv.
    1 focused subgoal
    (shelved: 1)

      P, Q : bool -> Prop
      PQequiv : forall a b : bool, P (a || b) <-> Q a
      a : bool
      ============================
      Q (~~ a)
```

thus in this case, the tactic `apply/PQequiv` is equivalent to `apply: (iffRL (PQequiv _ _))`, where `iffRL` is the analogue of `iffRL` for the converse implication.

Any SSReflect term whose type coerces to a double implication can be used as a view for goal interpretation.

Note that the goal interpretation view mechanism supports both `apply` and `exact` tactics. As expected, a goal interpretation view command exact/term should solve the current goal or it will fail.

Warning: Goal interpretation view tactics are *not* compatible with the bookkeeping tactical `=>` since this would be redundant with the `apply: term => _` construction.

Boolean reflection

In the Calculus of Inductive Constructions, there is an obvious distinction between logical propositions and boolean values. On the one hand, logical propositions are objects of *sort* `Prop` which is the carrier of intuitionistic reasoning. Logical connectives in `Prop` are *types*, which give precise information on the structure of their proofs; this information is automatically exploited by Coq tactics. For example, Coq knows that a proof of `A \/ B` is either a proof of `A` or a proof of `B`. The tactics `left` and `right` change the goal `A \/ B` to `A` and `B`, respectively; dually, the tactic `case` reduces the goal `A \/ B => G` to two subgoals `A => G` and `B => G`.

On the other hand, bool is an inductive *datatype* with two constructors true and false. Logical connectives on bool are *computable functions*, defined by their truth tables, using case analysis:

Example

```
Definition orb (b1 b2 : bool) := if b1 then true else b2.
    orb is defined
```

Properties of such connectives are also established using case analysis

Example

```
Lemma test b : b || ~~ b = true.
    1 subgoal

      b : bool
      ============================
      b || ~~ b = true

by case: b.
    No more subgoals.
```

Once `b` is replaced by `true` in the first goal and by `false` in the second one, the goals reduce by computations to the trivial `true = true`.

Thus, `Prop` and `bool` are truly complementary: the former supports robust natural deduction, the latter allows brute-force evaluation. SSReflect supplies a generic mechanism to have the best of the two worlds and move freely from a propositional version of a decidable predicate to its boolean version.

First, booleans are injected into propositions using the coercion mechanism:

```
Coercion is_true (b : bool) := b = true.
```

This allows any boolean formula `b` to be used in a context where Coq would expect a proposition, e.g., after `Lemma … :`. It is then interpreted as `(is_true b)`, i.e., the proposition `b = true`. Coercions are elided by the pretty-printer, so they are essentially transparent to the user.

The reflect predicate

To get all the benefits of the boolean reflection, it is in fact convenient to introduce the following inductive predicate `reflect` to relate propositions and booleans:

```
Inductive reflect (P: Prop): bool -> Type :=
| Reflect_true : P -> reflect P true
| Reflect_false : ~P -> reflect P false.
```

The statement `(reflect P b)` asserts that `(is_true b)` and `P` are logically equivalent propositions.

For instance, the following lemma:

```
Lemma andP: forall b1 b2, reflect (b1 /\ b2) (b1 && b2).
```

relates the boolean conjunction to the logical one `/\`. Note that in `andP`, `b1` and `b2` are two boolean variables and the proposition `b1 /\ b2` hides two coercions. The conjunction of `b1` and `b2` can then be viewed as `b1 /\ b2` or as `b1 && b2`.

Expressing logical equivalences through this family of inductive types makes possible to take benefit from *rewritable equations* associated to the case analysis of Coq's inductive types.

Since the equivalence predicate is defined in Coq as:

```
Definition iff (A B:Prop) := (A -> B) /\ (B -> A).
```

where `/\` is a notation for `and`:

```
Inductive and (A B:Prop) : Prop := conj : A -> B -> and A B.
```

This make case analysis very different according to the way an equivalence property has been defined.

```
Lemma andE (b1 b2 : bool) : (b1 /\ b2) <-> (b1 && b2).
```

Let us compare the respective behaviors of `andE` and `andP`.

Example

```
Lemma test (b1 b2 : bool) : if (b1 && b2) then b1 else ~~(b1||b2).
    1 subgoal

      b1, b2 : bool
      ============================
      if b1 && b2 then b1 else ~~ (b1 || b2)


case: (@andE b1 b2).
    1 subgoal

      b1, b2 : bool
      ============================
      (b1 /\ b2 -> b1 && b2) ->
      (b1 && b2 -> b1 /\ b2) -> if b1 && b2 then b1 else ~~ (b1 || b2)


case: (@andP b1 b2).
    2 subgoals

      b1, b2 : bool
```

(continues on next page)

(continued from previous page)

```
      ============================
      b1 /\ b2 -> b1

    subgoal 2 is:
     ~ (b1 /\ b2) -> ~~ (b1 || b2)
```

Expressing reflection relation through the `reflect` predicate is hence a very convenient way to deal with classical reasoning, by case analysis. Using the `reflect` predicate allows moreover to program rich specifications inside its two constructors, which will be automatically taken into account during destruction. This formalisation style gives far more efficient specifications than quantified (double) implications.

A naming convention in SSReflect is to postfix the name of view lemmas with `P`. For example, `orP` relates `||` and `\/`, `negP` relates `~~` and `~`.

The view mechanism is compatible with reflect predicates.

Example

```
Lemma test (a b : bool) (Ha : a) (Hb : b) : a /\ b.
    1 subgoal

      a, b : bool
      Ha : a
      Hb : b
      ============================
      a /\ b

apply/andP.
    1 focused subgoal
    (shelved: 1)

      a, b : bool
      Ha : a
      Hb : b
      ============================
      a && b
```

Conversely

```
Lemma test (a b : bool) : a /\ b -> a.
    1 subgoal

      a, b : bool
      ============================
      a /\ b -> a

move/andP.
    1 subgoal

      a, b : bool
      ============================
      a && b -> a
```

The same tactics can also be used to perform the converse operation, changing a boolean conjunction into a logical one. The view mechanism guesses the direction of the transformation to be used i.e., the constructor of the reflect predicate which should be chosen.

General mechanism for interpreting goals and assumptions

Specializing assumptions

The SSReflect tactic:

```
move/(_ term1 … termn).
```

is equivalent to the tactic:

```
intro top; generalize (top term1 … termn); clear top.
```

where `top` is a fresh name for introducing the top assumption of the current goal.

Interpreting assumptions

The general form of an assumption view tactic is:

Variant: `move | case / term`

The term , called the *view lemma* can be:

- a (term coercible to a) function;
- a (possibly quantified) implication;
- a (possibly quantified) double implication;
- a (possibly quantified) instance of the reflect predicate (see section *Views and reflection*).

Let `top` be the top assumption in the goal.

There are three steps in the behavior of an assumption view tactic:

- It first introduces `top`.
- If the type of *`term`* is neither a double implication nor an instance of the reflect predicate, then the tactic automatically generalises a term of the form: `term term1 … termn` where the terms `term1 … termn` instantiate the possible quantified variables of `term` , in order for `(term term1 … termn top)` to be well typed.
- If the type of `term` is an equivalence, or an instance of the reflect predicate, it generalises a term of the form: `(termvh (term term1 … termn ))` where the term `termvh` inserted is called an *assumption interpretation view hint*.
- It finally clears top.

For a `case/term` tactic, the generalisation step is replaced by a case analysis step.

View hints are declared by the user (see section *Views and reflection*) and are stored in the Hint View database. The proof engine automatically detects from the shape of the top assumption `top` and of the view lemma `term` provided to the tactic the appropriate view hint in the database to be inserted.

If `term` is a double implication, then the view hint will be one of the defined view hints for implication. These hints are by default the ones present in the file `ssreflect.v`:

```
Lemma iffLR : forall P Q, (P <-> Q) -> P -> Q.
```

which transforms a double implication into the left-to-right one, or:

```
Lemma iffRL : forall P Q, (P <-> Q) -> Q -> P.
```

which produces the converse implication. In both cases, the two first Prop arguments are implicit.

If `term` is an instance of the `reflect` predicate, then `A` will be one of the defined view hints for the `reflect` predicate, which are by default the ones present in the file `ssrbool.v`. These hints are not only used for choosing the appropriate direction of the translation, but they also allow complex transformation, involving negations.

Example

```
Check introN.
    introN
         : forall (P : Prop) (b : bool), reflect P b -> ~ P -> ~~ b

Lemma test (a b : bool) (Ha : a) (Hb : b) : ~~ (a && b).
    1 subgoal

      a, b : bool
      Ha : a
      Hb : b
      ============================
      ~~ (a && b)

apply/andP.
    1 focused subgoal
    (shelved: 1)

      a, b : bool
      Ha : a
      Hb : b
      ============================
      ~ (a /\ b)
```

In fact this last script does not exactly use the hint `introN`, but the more general hint:

```
Check introNTF.
    introNTF
         : forall (P : Prop) (b c : bool),
           reflect P b -> (if c then ~ P else P) -> ~~ b = c
```

The lemma `introN` is an instantiation of `introNF` using `c := true`.

Note that views, being part of *i_pattern*, can be used to interpret assertions too. For example the following script asserts `a && b` but actually uses its propositional interpretation.

Example

```
Lemma test (a b : bool) (pab : b && a) : b.
    1 subgoal

      a, b : bool
      pab : b && a
      ============================
```

(continues on next page)

(continued from previous page)

```
      b

have /andP [pa ->] : (a && b) by rewrite andbC.
    1 subgoal

      a, b : bool
      pab : b && a
      pa : a
      ============================
      true
```

Interpreting goals

A goal interpretation view tactic of the form:

Variant: `apply/`*`term`*

applied to a goal `top` is interpreted in the following way:

- If the type of `term` is not an instance of the `reflect` predicate, nor an equivalence, then the term `term` is applied to the current goal `top`, possibly inserting implicit arguments.
- If the type of `term` is an instance of the reflect predicate or an equivalence, then a *goal interpretation view hint* can possibly be inserted, which corresponds to the application of a term `(termvh (term _ … _))` to the current goal, possibly inserting implicit arguments.

Like assumption interpretation view hints, goal interpretation ones are user-defined lemmas stored (see section *Views and reflection*) in the `Hint View` database bridging the possible gap between the type of `term` and the type of the goal.

Interpreting equivalences

Equivalent boolean propositions are simply *equal* boolean terms. A special construction helps the user to prove boolean equalities by considering them as logical double implications (between their coerced versions), while performing at the same time logical operations on both sides.

The syntax of double views is:

Variant: `apply/`*`term`*`/`*`term`*

The first term is the view lemma applied to the left hand side of the equality, while the second term is the one applied to the right hand side.

In this context, the identity view can be used when no view has to be applied:

```
Lemma idP : reflect b1 b1.
```

Example

```
Lemma test (b1 b2 b3 : bool) : ~~ (b1 || b2) = b3.
    1 subgoal

      b1, b2, b3 : bool
      ============================
      ~~ (b1 || b2) = b3
```

(continues on next page)

(continued from previous page)

```
apply/idP/idP.
    2 focused subgoals
    (shelved: 2)

      b1, b2, b3 : bool
      ============================
      ~~ (b1 || b2) -> b3

    subgoal 2 is:
     b3 -> ~~ (b1 || b2)
```

The same goal can be decomposed in several ways, and the user may choose the most convenient interpretation.

```
Lemma test (b1 b2 b3 : bool) : ~~ (b1 || b2) = b3.
    1 subgoal

      b1, b2, b3 : bool
      ============================
      ~~ (b1 || b2) = b3

apply/norP/idP.
    2 focused subgoals
    (shelved: 2)

      b1, b2, b3 : bool
      ============================
      ~~ b1 /\ ~~ b2 -> b3

    subgoal 2 is:
     b3 -> ~~ b1 /\ ~~ b2
```

Declaring new Hint Views

Command: `Hint View for move /` *ident* `|` *num* ?

Command: `Hint View for apply /` *ident* `|` *num* ?

This command can be used to extend the database of hints for the view mechanism.

As library `ssrbool.v` already declares a corpus of hints, this feature is probably useful only for users who define their own logical connectives.

The *ident* is the name of the lemma to be declared as a hint. If `move` is used as tactic, the hint is declared for assumption interpretation tactics, `apply` declares hints for goal interpretations. Goal interpretation view hints are declared for both simple views and left hand side views. The optional natural number is the number of implicit arguments to be considered for the declared hint view lemma.

Variant: `Hint View for apply//`*ident* `|` *num* ?

This variant with a double slash `//`, declares hint views for right hand sides of double views.

See the files `ssreflect.v` and `ssrbool.v` for examples.

Multiple views

The hypotheses and the goal can be interpreted by applying multiple views in sequence. Both move and apply can be followed by an arbitrary number of `/term`. The main difference between the following two tactics

```
apply/v1/v2/v3.
apply/v1; apply/v2; apply/v3.
```

is that the former applies all the views to the principal goal. Applying a view with hypotheses generates new goals, and the second line would apply the view `v2` to all the goals generated by `apply/v1`.

Note that the NO-OP intro pattern `-` can be used to separate two views, making the two following examples equivalent:

```
move=> /v1; move=> /v2.
move=> /v1 - /v2.
```

The tactic `move` can be used together with the `in` tactical to pass a given hypothesis to a lemma.

Example

```
Variable P2Q : P -> Q.
    P2Q is declared

Variable Q2R : Q -> R.
    Q2R is declared

Lemma test (p : P) : True.
    1 subgoal

      P, Q, R : Prop
      P2Q : P -> Q
      Q2R : Q -> R
      p : P
      ============================
      True

move/P2Q/Q2R in p.
    1 subgoal

      P, Q, R : Prop
      P2Q : P -> Q
      Q2R : Q -> R
      p : R
      ============================
      True
```

If the list of views is of length two, `Hint Views` for interpreting equivalences are indeed taken into account, otherwise only single `Hint Views` are used.

5.7.9 SSReflect searching tool

Command: `Search` *pattern*? [-? [*string* | *pattern*] [% *ident*]?]* [`in` [-? *qualid*]+]?

This is the SSReflect extension of the Search command. *qualid* is the name of an open module. This command returns the list of lemmas:

- whose *conclusion* contains a subterm matching the optional first pattern. A `-` reverses the test, producing the list of lemmas whose conclusion does not contain any subterm matching the pattern;
- whose name contains the given string. A `-` prefix reverses the test, producing the list of lemmas whose name does not contain the string. A string that contains symbols or is followed by a scope key, is interpreted as the constant whose notation involves that string (e.g., `+` for `addn`), if this is unambiguous; otherwise the diagnostic includes the output of the *Locate* vernacular command.
- whose statement, including assumptions and types, contains a subterm matching the next patterns. If a pattern is prefixed by `-`, the test is reversed;
- contained in the given list of modules, except the ones in the modules prefixed by a `-`.

Note:

- As for regular terms, patterns can feature scope indications. For instance, the command: `Search _ (_ + _)%N.` lists all the lemmas whose statement (conclusion or hypotheses) involves an application of the binary operation denoted by the infix `+` symbol in the `N` scope (which is SSReflect scope for natural numbers).
- Patterns with holes should be surrounded by parentheses.
- Search always volunteers the expansion of the notation, avoiding the need to execute Locate independently. Moreover, a string fragment looks for any notation that contains fragment as a substring. If the `ssrbool.v` library is imported, the command: `Search "~~".` answers :

```
Search "~~".
    "~~" is part of notation ("~~ _")
    In bool_scope, ("~~ b") denotes negb b
    Toplevel input, characters 0-12:
    > Search "~~".
    > ^^^^^^^^^^^^
    Warning: Listing only lemmas with conclusion matching (~~ ?b)
    negbT: forall b : bool, b = false -> ~~ b

    contra: forall c b : bool, (c -> b) -> ~~ b -> ~~ c

    contraNN: forall c b : bool, (c -> b) -> ~~ b -> ~~ c

    contraL: forall c b : bool, (c -> ~~ b) -> b -> ~~ c

    contraTN: forall c b : bool, (c -> ~~ b) -> b -> ~~ c

    contraFN: forall c b : bool, (c -> b) -> b = false -> ~~ c

    introN: forall (P : Prop) (b : bool), reflect P b -> ~ P -> ~~ b
```

- A diagnostic is issued if there are different matching notations; it is an error if all matches are partial.
- Similarly, a diagnostic warns about multiple interpretations, and signals an error if there is no default one.

- The command `Search in M.` is a way of obtaining the complete signature of the module `M`.
- Strings and pattern indications can be interleaved, but the first indication has a special status if it is a pattern, and only filters the conclusion of lemmas:
  - The command : `Search (_ =1 _) "bij".` lists all the lemmas whose conclusion features a `=1` and whose name contains the string `bij`.
  - The command : `Search "bij" (_ =1 _).` lists all the lemmas whose statement, including hypotheses, features a `=1` and whose name contains the string `bij`.

5.7.10 Synopsis and Index

Parameters

SSReflect tactics

d_tactic ::= elim | case | congr | apply | exact | move

Notation scope

key ::= *ident*

Module name

modname ::= *qualid*

Natural number

natural ::= *num* | *ident*

where *ident* is an Ltac variable denoting a standard Coq numeral (should not be the name of a tactic which can be followed by a bracket `[`, like `do`, `have`,…)

Items and switches

ssr_binder ::= *ident* | (*ident* [: *term*]?)

binder see *Abbreviations*.

clear_switch ::= { [*ident*]+ }

clear switch see *Discharge*

c_pattern ::= [*term* in | *term* as]? *ident* in *term*

context pattern see *Contextual patterns*

d_item ::= [*occ_switch* | *clear_switch*]? [*term* | (*c_pattern*)]?

discharge item see *Discharge*

gen_item ::= [@]? *ident* | (*ident*) | ([@]? *ident* := *c_pattern*)

generalization item see *Structure*

i_pattern ::= *ident* | > | _ | ? | * | + | [*occ_switch*]? [-> | <-] | [[*i_item*]? [|]] | - | [: *iden* intro pattern *Introduction in the context*

i_item ::= *clear_switch* | *s_item* | *i_pattern* | *i_view* | *i_block*

view *Introduction in the context*

i_view ::= `{}`? `/`*term* | `/ltac:(` *tactic* `)`

intro block *Introduction in the context*

i_block ::= `[^` *ident* `]` | `[^~` *ident* | *num* `]`

intro item see *Introduction in the context*

int_mult ::= *num*? *mult_mark*

multiplier see *Iteration*

occ_switch ::= `{` `+` | `-`? *num** `}`

occur. switch see *Occurrence selection*

mult ::= *num*? *mult_mark*

multiplier see *Iteration*

mult_mark ::= `?` | `!`

multiplier mark see *Iteration*

r_item ::= `/`? *term* | *s_item*

rewrite item see *Rewriting*

r_prefix ::= `-`? *int_mult*? *occ_switch* | *clear_switch*? `[` *r_pattern* `]`?

rewrite prefix see *Rewriting*

r_pattern ::= *term* | *c_pattern* | `in` *ident* `in`? *term*

rewrite pattern see *Rewriting*

r_step ::= *r_prefix*? *r_item*

rewrite step see *Rewriting*

s_item ::= `/=` | `//` | `//=`

simplify switch see *Introduction in the context*

Tactics

Note: `without loss` and `suffices` are synonyms for `wlog` and `suff` respectively.

`move`

idtac or *hnf* (see *Bookkeeping*)

`apply`
`exact`

application (see *The defective tactics*)

`abstract`

see *The abstract tactic* and *Generating let in context entries with have* `elim`

induction (see *The defective tactics*)

`case`

case analysis (see *The defective tactics*)

`rewrite r_step+`

rewrite (see *Rewriting*)

`under r_prefix? term (=> i_item+)? (do (tactic | [ tactic*| ]))?`

under (see *Rewriting under binders*)

`over`

over (see *The over tactic*)

`have i_item* i_pattern? (s_item | ssr_binder+)? (: term)? := term`

`have i_item* i_pattern? (s_item | ssr_binder+)? : term (by tactic)?`

`have suff clear_switch? i_pattern? (: term)? := term`

`have suff clear_switch? i_pattern? : term (by tactic)?`

`gen have (ident ,)? i_pattern? : gen_item+ / term (by tactic)?`

`generally have (ident ,)? i_pattern? : gen_item+ / term (by tactic)?`

forward chaining (see *Structure*)

`wlog suff? i_item? : (gen_item | clear_switch)* / term`

specializing (see *Structure*)

`suff i_item* i_pattern? ssr_binder+ : term (by tactic)?`

`suffices i_item* i_pattern? ssr_binder+ : term (by tactic)?`

`suff have? clear_switch? i_pattern? : term (by tactic)?`

`suffices have? clear_switch? i_pattern? : term (by tactic)?`

backchaining (see *Structure*)

`pose ident := term`

local definition (see *Definitions*)

Variant: `pose ident ssr_binder+ := term`

local function definition

Variant: `pose fix fix_body`

local fix definition

Variant: `pose cofix fix_body`

local cofix definition

`set ident (: term)? := occ_switch? (term | ( c_pattern))`

abbreviation (see *Abbreviations*)

`unlock ( r_prefix? ident )*`

unlock (see *Locking, unlocking*)

`congr num? term`

congruence (see *Congruence*)

Tacticals

`tactic += d_tactic ident? : d_item+ clear_switch?`

discharge *Discharge*

`tactic += tactic => i_item+`

introduction see *Introduction in the context*

`tactic += tactic in ( gen_item | clear_switch )+ *?`

localization see *Localization*

`tactic += do mult? ( tactic | [ tactic+| ] )`

iteration see *Iteration*

`tactic += tactic ; ( first | last ) num? ( tactic | [ tactic+| ] )`

selector see *Selectors*

`tactic += tactic ; ( first | last ) num?`

rotation see *Selectors*

`tactic += by ( tactic | [ tactic*| ] )`

closing see *Terminators*

Commands

Command: `Hint View for ( move | apply ) / ident ( | num )?`

view hint declaration (see *Declaring new Hint Views*)

Command: `Hint View for apply // ident num?`

right hand side double , view hint declaration (see *Declaring new Hint Views*)

Command: `Prenex Implicits ident+`

prenex implicits declaration (see *Parametric polymorphism*)

Settings

Flag: `Debug Ssreflect`

Developer only. Print debug information on reflect.

Flag: `Debug SsrMatching`

Developer only. Print debug information on SSR matching.

CHAPTER

SIX

USER EXTENSIONS

6.1 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 to provide custom symbolic notations for terms are *Notation* and *Infix*; they will be described in the *next section*. There is also a variant of *Notation* which does not modify the parser; this provides a form of *abbreviation*. It is sometimes expected that the same symbolic notation has different meanings in different contexts; to achieve this form of overloading, Coq offers a notion of *interpretation scopes*. The main command to provide custom notations for tactics is *Tactic Notation*.

6.1.1 Notations

Basic notations

Command: `Notation`

A *notation* is a symbolic expression 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

```
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 (except when the abbreviation has the form of an ordinary applicative expression; see *Abbreviations*). The notation is composed of *tokens* separated by spaces. Identifiers in the string (such as `A` and `B`) are the *parameters* of the notation. Each of them must occur at least once 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 single 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.

```
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 *Reserving notations*), explicit precedences and associativity rules have to be given.

Note: The right-hand side of a notation is interpreted at the time the notation is given. In particular, disambiguation of constants, *implicit arguments* and other notations are resolved at the time of the declaration of the notation.

Precedences and associativity

Mixing different symbolic notations in the 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

```
Notation "A \/ B" := (or A B).
```

Clearly, an expression such as `forall A:Prop, True /\ A \/ A \/ False` is ambiguous. To tell the Coq parser how to interpret the expression, a priority between the symbols `/\` and `\/` has to be given. Assume for instance that we want conjunction to bind more than disjunction. This is expressed by assigning a precedence level to each notation, knowing that a lower level binds more than a higher level. Hence the level for disjunction must be higher than the level for conjunction.

Since connectives are not tight articulation points of a text, it is reasonable to choose levels not so far from the highest level which is 100, for example 85 for disjunction and 80 for conjunction[308].

Similarly, an associativity is needed to decide whether `True /\ False /\ False` defaults to `True /\ (False /\ False)` (right associativity) or to `(True /\ False) /\ False` (left associativity). We may even consider that the expression is not well-formed and that parentheses are mandatory (this is a "no associativity")[309]. We do not know of a special convention of the associativity of disjunction and conjunction, so let us 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.

```
Notation "A /\ B" := (and A B) (at level 80, right associativity).
Notation "A \/ B" := (or A B) (at level 85, right associativity).
```

By default, a notation is considered nonassociative, but the precedence level is mandatory (except for special cases whose level is canonical). The level is either a number or the phrase `next level` whose meaning is obvious. Some *associativities are predefined* in the `Notations` module.

Complex notations

Notations can be made from arbitrarily complex symbols. One can for instance define prefix notations.

```
Notation "~ x" := (not x) (at level 75, right associativity).
```

One can also define notations for incomplete terms, with the hole expected to be inferred during type checking.

```
Notation "x = y" := (@eq _ x y) (at level 70, no associativity).
```

[308] which are the levels effectively chosen in the current implementation of Coq

[309] Coq accepts notations declared as nonassociative but the parser on which Coq is built, namely Camlp5, currently does not implement `no associativity` and replaces it with `left associativity`; hence it is the same for Coq: `no associativity` is in fact `left associativity` for the purposes of parsing

One can define *closed* notations whose both sides are symbols. In this case, the default precedence level for the inner sub-expression is 200, and the default level for the notation itself is 0.

```
Notation "( x , y )" := (@pair _ _ x y).
```

One can also define notations for binders.

```
Notation "{ x : A | P }" := (sig A (fun x => P)).
```

In the last case though, there is a conflict with the notation for type casts. The notation for types casts, 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 the following:

```
Notation "{ x : A | P }" := (sig A (fun x => P)) (x at level 99).
    Setting notation at level 0.
```

More generally, it is required that notations are explicitly factorized on the left. See the next section for more about factorization.

Simple factorization rules

Coq extensible parsing is performed by *Camlp5* which is essentially a LL1 parser: it decides which notation to parse by looking at tokens from left to right. 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.

```
Notation "x < y" := (lt x y) (at level 70).
Fail Notation "x < y < z" := (x < y /\ y < z) (at level 70).
    The command has indeed failed with message:
    Notation "_ < _ < _" is already defined at level 70 with arguments constr
    at next level, constr at next level, constr at next level
    while it is now required to be at level 70 with arguments constr
    at next level, constr Unknown level, constr at next level.
```

In order to factorize the left part of the rules, the subexpression referred to 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 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.

```
Notation "x < y" := (lt x y) (at level 70).
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 the chapter on *The Coq library*.

Command: **`Print Grammar constr.`**

This command displays the current state of the Coq term parser.

Command: **`Print Grammar pattern.`**

This displays the state of the subparser of patterns (the parser used in the grammar of the `match with` constructions).

Displaying symbolic notations

The command *`Notation`* has an effect both on the Coq parser and on the Coq printer. For example:

```
Check (and True True).
    True /\ True
         : Prop
```

However, printing, especially pretty-printing, also requires some care. We may want specific indentations, line breaks, alignment if on several lines, etc. For pretty-printing, Coq relies on OCaml formatting library, which provides indentation and automatic line breaks depending on page width by means of *formatting boxes*.

The default printing of notations is 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.

```
Notation "{{ x : A | P }}" := (sig (fun x : A => P)) (at level 0, x at level 99).


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

```
Notation "'If' c1 'then' c2 'else' c3" := (IF_then_else c1 c2 c3)
(at level 200, right associativity, format
"'[v   ' 'If'  c1 '/' '[' 'then'  c2  ']' '/' '[' 'else'  c3 ']' ']'").
    Identifier 'If' now a keyword


Check
  (IF_then_else (IF_then_else True False True)
    (IF_then_else True False True)
    (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
```

A *format* is an extension of the string denoting the notation with the possible following elements delimited by single quotes:

- tokens of the form `'/  '` are translated into breaking points. If there is a line break, indents the number of spaces appearing after the "`/`" (no indentation in the example)
- tokens of the form `'//'` force writing on a new line
- well-bracketed pairs of tokens of the form `'[  '` and `']'` are translated into printing boxes; if there is a line break, an extra indentation of the number of spaces after the "`[`" is applied
- 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

new line and an extra indentation of the number of spaces after the "`[hv`" is applied at the beginning of each new line

- well-bracketed pairs of tokens of the form `'[v '` and `']'` are translated into vertical printing boxes; every breaking point forces a new line, even if the line is large enough to display the whole content of the box, and an extra indentation of the number of spaces after the "`[v`" is applied at the beginning of each new line (3 spaces in the example)
- extra spaces in other tokens are preserved in the output

Notations disappear when a section is closed. No typing of the denoted expression is performed at definition time. Type checking is done only at the time of use of the notation.

Note: 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*. Conversely, the `only printing` *modifier* can be used to declare that a notation should only be used for printing and should not declare a parsing rule. In particular, such notations do not modify the parser.

The Infix command

The *Infix* command is a shortening for declaring notations of infix symbols.

Command: `Infix "`*symbol*`" :=` *term* `(`*modifiers*`)`? `.`

This command is equivalent to

```
Notation "x symbol y" := (term x y) (modifiers)? .
```

where `x` and `y` are fresh names. Here is an example.

```
Infix "/\" := and (at level 80, right associativity).
```

Reserving notations

Command: `Reserved Notation` *string* `(`*modifiers*`)`?

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, this command declares a parsing rule (*string*) in advance without giving its interpretation. Here is an example from the initial state of Coq.

```
Reserved Notation "x = y" (at level 70, no associativity).
```

Reserving a notation is also useful for simultaneously defining an inductive type or a recursive constant and a notation for it.

Note: The notations mentioned in the module *Notations* are reserved. Hence their precedence and associativity cannot be changed.

Variant: `Reserved Infix "`*symbol*`"` *modifiers**

This command declares an infix parsing rule without giving its interpretation.

Simultaneous definition of terms and notations

Thanks to reserved notations, the inductive, co-inductive, record, recursive and corecursive definitions can benefit from 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 by *`decl_notation`* for inductive, co-inductive, recursive and corecursive definitions and in *Record types* for records. Here are examples:

```
Reserved Notation "A & B" (at level 80).
```

```
Inductive and' (A B : Prop) : Prop := conj' : A -> B -> A & B
where "A & B" := (and' A B).
```

```
Fixpoint plus (n m : nat) {struct n} : nat :=
match n with
    | O => m
    | S p => S (p+m)
end
where "n + m" := (plus n m).
```

Displaying information about notations

Flag: `Printing Notations`

Controls whether to use notations for printing terms wherever possible. Default is on.

See also:

`Printing All` To disable other elements in addition to notations.

Locating notations

To know to which notations a given symbol belongs to, use the *`Locate`* command. You can call it on any (composite) symbol surrounded by double quotes. To locate a particular notation, use a string where the variables of the notation are replaced by “_” and where possible single quotes inserted around identifiers or tokens starting with a single quote are dropped.

```
Locate "exists".
    Notation
    "'exists' x .. y , p" := ex (fun x => .. (ex (fun y => p)) ..) : type_scope
    (default interpretation)
    "'exists' ! x .. y , p" := ex
                                 (unique
                                    (fun x => .. (ex (unique (fun y => p))) ..))
    : type_scope (default interpretation)
```

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

Notations and binders

Notations can include binders. This section lists different ways to deal with binders. For further examples, see also *Notations with recursive patterns involving binders*.

Binders bound in the notation and parsed as identifiers

Here is the basic example of a notation using a binder:

```
Notation "'sigma' x : A , B" := (sigT (fun x : A => B))
  (at level 200, x ident, A at level 200, right associativity).
```

The binding variables in the right-hand side that occur as a parameter of the notation (here `x`) dynamically bind all the occurrences in their respective binding scope after instantiation of the parameters of the notation. This means that the term bound to `B` can refer to the variable name bound to `x` as shown in the following application of the notation:

```
Check sigma z : nat, z = 0.
    sigma z : nat, z = 0
         : Set
```

Notice the *modifier* `x ident` in the declaration of the notation. It tells to parse `x` as a single identifier.

Binders bound in the notation and parsed as patterns

In the same way as patterns can be used as binders, as in `fun '(x,y) => x+y` or `fun '(existT _ x _) => x`, notations can be defined so that any *pattern* can be used in place of the binder. Here is an example:

```
Notation "'subset' ' p , P " := (sig (fun p => P))
  (at level 200, p pattern, format "'subset'  ' p ,  P").


Check subset '(x,y), x+y=0.
    subset '(x, y), x + y = 0
         : Set
```

The *modifier* `p pattern` in the declaration of the notation tells to parse `p` as a pattern. Note that a single variable is both an identifier and a pattern, so, e.g., the following also works:

```
Check subset 'x, x=0.
    subset 'x, x = 0
         : Set
```

If one wants to prevent such a notation to be used for printing when the pattern is reduced to a single identifier, one has to use instead the *modifier* `p strict pattern`. For parsing, however, a `strict pattern` will continue to include the case of a variable. Here is an example showing the difference:

```
Notation "'subset_bis' ' p , P" := (sig (fun p => P))
  (at level 200, p strict pattern).
Notation "'subset_bis' p , P " := (sig (fun p => P))
  (at level 200, p ident).


Check subset_bis 'x, x=0.
    subset_bis x, x = 0
         : Set
```

The default level for a `pattern` is 0. One can use a different level by using `pattern at level` n where the scale is the same as the one for terms (see *Notations*).

Binders bound in the notation and parsed as terms

Sometimes, for the sake of factorization of rules, a binder has to be parsed as a term. This is typically the case for a notation such as the following:

```
Notation "{ x : A | P }" := (sig (fun x : A => P))
    (at level 0, x at level 99 as ident).
```

This is so because the grammar also contains rules starting with `{}` and followed by a term, such as the rule for the notation `{ A } + { B }` for the constant `sumbool` (see *Specification*).

Then, in the rule, `x ident` is replaced by `x at level 99 as ident` meaning that `x` is parsed as a term at level 99 (as done in the notation for `sumbool`), but that this term has actually to be an identifier.

The notation `{ x | P }` is already defined in the standard library with the `as ident` *`modifier`*. We cannot redefine it but one can define an alternative notation, say `{ p such that P }`, using instead `as pattern`.

```
Notation "{ p 'such' 'that' P }" := (sig (fun p => P))
  (at level 0, p at level 99 as pattern).
```

Then, the following works:

```
Check {(x,y) such that x+y=0}.
    {(x, y) such that x + y = 0}
         : Set
```

To enforce that the pattern should not be used for printing when it is just an identifier, one could have said `p at level 99 as strict pattern`.

Note also that in the absence of a `as ident`, `as strict pattern` or `as pattern` *`modifier`*s, the default is to consider sub-expressions occurring in binding position and parsed as terms to be `as ident`.

Binders not bound in the notation

We can also have binders in the right-hand side of a notation which are not themselves bound in the notation. In this case, the binders are considered up to renaming of the internal binder. E.g., for the notation

```
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.

```
Fail Check (exists_different p).
    The command has indeed failed with message:
    The reference p was not found in the current environment.
```

```
Notation "[> a , .. , b <]" :=
  (cons a .. (cons b nil) .., cons b .. (cons a nil) ..).
```

Notations with recursive patterns

A mechanism is provided for declaring elementary notations with recursive patterns. The basic example is:

```
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).
    Setting notation at level 0.
```

On the right-hand side, an extra construction of the form `.. t ..` can be used. Notice that `..` is part of the Coq syntax and it must not be confused with the three-dots notation "…" used in this manual to denote a sequence of arbitrary size.

On the left-hand side, the part "`x s .. s y`" of the notation parses any number of times (but at least once) a sequence of expressions separated by the sequence of tokens `s` (in the example, `s` is just "`;`").

The right-hand side must contain a subterm of the form either ϕ`(x, ..` ϕ`(y,t) ..)` or ϕ`(y, ..` ϕ`(x,t) ..)` where $\varphi([~]_E,[~]_I)$, called the *iterator* of the recursive notation is an arbitrary expression with distinguished placeholders and where t is called the *terminating expression* of the recursive notation. In the example, we choose the names x and y but in practice they can of course be chosen arbitrarily. Note that the placeholder $[~]_I$ has to occur only once but $[~]_E$ can occur several times.

Parsing the notation produces a list of expressions which are used to fill the first placeholder of the iterating pattern which itself is repeatedly nested as many times as the length of the list, the second placeholder being the nesting point. In the innermost occurrence of the nested iterating pattern, the second placeholder is finally filled with the terminating expression.

In the example above, the iterator $\varphi([~]_E,[~]_I)$ is $cons[~]_E\,[~]_I$ and the terminating expression is `nil`. Here are other examples:

```
Notation "( x , y , .. , z )" := (pair .. (pair x y) .. z) (at level 0).
Notation "[| t * ( x , y , .. , z ) ; ( a , b , .. , c )  * u |]" :=
  (pair (pair .. (pair (pair t x) (pair t y)) .. (pair t z))
        (pair .. (pair (pair a u) (pair b u)) .. (pair c u)))
  (t at level 39).
```

Notations with recursive patterns can be reserved like standard notations, they can also be declared within *interpretation scopes*.

Notations with recursive patterns involving binders

Recursive notations can also be used with binders. The basic example is:

```
Notation "'exists' x .. y , p" :=
  (ex (fun x => .. (ex (fun y => p)) ..))
  (at level 200, x binder, y binder, right associativity).
```

The principle is the same as in *Notations with recursive patterns* except that in the iterator $\varphi([~]_E,[~]_I)$, the placeholder $[~]_E$ can also occur in position of the binding variable of a `fun` or a `forall`.

To specify that the part "`x .. y`" of the notation parses a sequence of binders, `x` and `y` must be marked as `binder` in the list of *`modifiers`* of the notation. The binders of the parsed sequence are used to fill the occurrences of the first placeholder of the iterating pattern which is repeatedly nested as many times as the number of binders generated. If ever the generalization operator `'` (see *Implicit generalization*) is used in the binding list, the added binders are taken into account too.

There are two flavors of binder parsing. If `x` and `y` are marked as binder, then a sequence such as `a b c : T` will be accepted and interpreted as the sequence of binders `(a:T) (b:T) (c:T)`. For instance, in the notation above, the syntax `exists a b : nat, a = b` is valid.

The variables `x` and `y` can also be marked as closed binder in which case only well-bracketed binders of the form `(a b c:T)` or `{a b c:T}` etc. are accepted.

With closed binders, the recursive sequence in the left-hand side can be of the more general form `x s .. s y` where `s` is an arbitrary sequence of tokens. With open binders though, `s` has to be empty. Here is an example of recursive notation with closed binders:

```
Notation "'mylet' f x .. y :=  t 'in' u":=
  (let f := fun x => .. (fun y => t) .. in u)
  (at level 200, x closed binder, y closed binder, right associativity).
```

A recursive pattern for binders can be used in position of a recursive pattern for terms. Here is an example:

```
Notation "'FUNAPP' x .. y , f" :=
  (fun x => .. (fun y => (.. (f x) ..) y ) ..)
  (at level 200, x binder, y binder, right associativity).
```

If an occurrence of the $[\ ]_E$ is not in position of a binding variable but of a term, it is the name used in the binding which is used. Here is an example:

```
Notation "'exists_non_null' x .. y  , P" :=
  (ex (fun x => x <> 0 /\ .. (ex (fun y => y <> 0 /\ P)) ..))
  (at level 200, x binder).
```

Predefined entries

By default, sub-expressions are parsed as terms and the corresponding grammar entry is called `constr`. However, one may sometimes want to restrict the syntax of terms in a notation. For instance, the following notation will accept to parse only global reference in position of `x`:

```
Notation "'apply' f a1 .. an" := (.. (f a1) .. an)
  (at level 10, f global, a1, an at level 9).
```

In addition to `global`, one can restrict the syntax of a sub-expression by using the entry names `ident` or `pattern` already seen in *Binders not bound in the notation*, even when the corresponding expression is not used as a binder in the right-hand side. E.g.:

```
Notation "'apply_id' f a1 .. an" := (.. (f a1) .. an)
  (at level 10, f ident, a1, an at level 9).
```

Custom entries

Command: `Declare Custom Entry` *`ident`*

This command allows to define new grammar entries, called *custom entries*, that can later be referred to using the entry name `custom` *`ident`*.

Example

For instance, we may want to define an ad hoc parser for arithmetical operations and proceed as follows:

```
Inductive Expr :=
| One : Expr
| Mul : Expr -> Expr -> Expr
| Add : Expr -> Expr -> Expr.
    Expr is defined
    Expr_rect is defined
    Expr_ind is defined
    Expr_rec is defined
    Expr_sind is defined
```

(continues on next page)

(continued from previous page)

```
Declare Custom Entry expr.
Notation "[ e ]" := e (e custom expr at level 2).
    Setting notation at level 0.

Notation "1" := One (in custom expr at level 0).
Notation "x y" := (Mul x y) (in custom expr at level 1, left associativity).
Notation "x + y" := (Add x y) (in custom expr at level 2, left associativity).
Notation "( x )" := x (in custom expr, x at level 2).
    Setting notation at level 0.

Notation "{ x }" := x (in custom expr, x constr).
    Setting notation at level 0.

Notation "x" := x (in custom expr at level 0, x ident).
Axiom f : nat -> Expr.
    f is declared

Check fun x y z => [1 + y z + {f x}].
    fun (x : nat) (y z : Expr) => [1 + y z + {f x}]
         : nat -> Expr -> Expr -> Expr

Unset Printing Notations.
Check fun x y z => [1 + y z + {f x}].
    fun (x : nat) (y z : Expr) => Add (Add One (Mul y z)) (f x)
         : forall (_ : nat) (_ : Expr) (_ : Expr), Expr

Set Printing Notations.
Check fun e => match e with
| [1 + 1] => [1]
| [x y + z] => [x + y z]
| y => [y + e]
end.
    fun e : Expr =>
    match e with
    | [1 + 1] => [1]
    | [x y + z] => [x + y z]
    | _ => [e + e]
    end
         : Expr -> Expr
```

Custom entries have levels, like the main grammar of terms and grammar of patterns have. The lower level is 0 and this is the level used by default to put rules delimited with tokens on both ends. The level is left to be inferred by Coq when using `in custom` *ident*. The level is otherwise given explicitly by using the syntax `in custom` *ident* `at level` *num*, where *num* refers to the level.

Levels are cumulative: a notation at level `n` of which the left end is a term shall use rules at level less than `n` to parse this subterm. More precisely, it shall use rules at level strictly less than `n` if the rule is declared with `right associativity` and rules at level less or equal than `n` if the rule is declared with `left associativity`. Similarly, a notation at level `n` of which the right end is a term shall use by default rules at level strictly less than `n` to parse this subterm if the rule is declared left associative and rules at level less or equal than `n` if the rule is declared right associative. This is what happens for instance in the rule

```
Notation "x + y" := (Add x y) (in custom expr at level 2, left associativity).
```

where `x` is any expression parsed in entry `expr` at level less or equal than `2` (including, recursively, the given rule) and `y` is any expression parsed in entry `expr` at level strictly less than `2`.

Rules associated to an entry can refer different sub-entries. The grammar entry name `constr` can be used to refer to the main grammar of term as in the rule

```
Notation "{ x }" := x (in custom expr at level 0, x constr).
```

which indicates that the subterm `x` should be parsed using the main grammar. If not indicated, the level is computed as for notations in `constr`, e.g. using 200 as default level for inner sub-expressions. The level can otherwise be indicated explicitly by using `constr at level n` for some `n`, or `constr at next level`.

Conversely, custom entries can be used to parse sub-expressions of the main grammar, or from another custom entry as is the case in

```
Notation "[ e ]" := e (e custom expr at level 2).
```

to indicate that `e` has to be parsed at level `2` of the grammar associated to the custom entry `expr`. The level can be omitted, as in

```
Notation "[ e ]" := e (e custom expr).
```

in which case Coq infer it. If the sub-expression is at a border of the notation (as e.g. `x` and `y` in `x + y`), the level is determined by the associativity. If the sub-expression is not at the border of the notation (as e.g. `e` in `"[ e ]`), the level is inferred to be the highest level used for the entry. In particular, this level depends on the highest level existing in the entry at the time of use of the notation.

In the absence of an explicit entry for parsing or printing a sub-expression of a notation in a custom entry, the default is to consider that this sub-expression is parsed or printed in the same custom entry where the notation is defined. In particular, if `x at level n` is used for a sub-expression of a notation defined in custom entry `foo`, it shall be understood the same as `x custom foo at level n`.

In general, rules are required to be *productive* on the right-hand side, i.e. that they are bound to an expression which is not reduced to a single variable. If the rule is not productive on the right-hand side, as it is the case above for

```
Notation "( x )" := x (in custom expr at level 0, x at level 2).
```

and

```
Notation "{ x }" := x (in custom expr at level 0, x constr).
```

it is used as a *grammar coercion* which means that it is used to parse or print an expression which is not available in the current grammar at the current level of parsing or printing for this grammar but which is available in another grammar or in another level of the current grammar. For instance,

```
Notation "( x )" := x (in custom expr at level 0, x at level 2).
```

tells that parentheses can be inserted to parse or print an expression declared at level `2` of `expr` whenever this expression is expected to be used as a subterm at level 0 or 1. This allows for instance to parse and print `Add x y` as a subterm of `Mul (Add x y) z` using the syntax `(x + y) z`. Similarly,

```
Notation "{ x }" := x (in custom expr at level 0, x constr).
```

gives a way to let any arbitrary expression which is not handled by the custom entry `expr` be parsed or printed by the main grammar of term up to the insertion of a pair of curly brackets.

Command: `Print Custom Grammar` *ident*.

This displays the state of the grammar for terms associated to the custom entry *ident*.

Summary

Syntax of notations

The different syntactic forms taken by the commands declaring notations are given below. The optional `scope` is described in *Interpretation scopes*.

```
notation          ::=  [Local] Notation string := term [(modifiers)] [: scope].
                       [Local] Infix string := qualid [(modifiers)] [: scope].
                       [Local] Reserved Notation string [(modifiers)] .
                       Inductive ind_body [decl_notation] with … with ind_body [decl_notation].
                       CoInductive ind_body [decl_notation] with … with ind_body [decl_notation].
                       Fixpoint fix_body [decl_notation] with … with fix_body [decl_notation].
                       CoFixpoint cofix_body [decl_notation] with … with cofix_body [decl_notation].
                       [Local] Declare Custom Entry ident.
decl_notation     ::=  [where string := term [: scope] and … and string := term [: scope]].
modifiers         ::=  modifier, … , modifier
modifier          ::=  at level num
                       in custom ident
                       in custom ident at level num
                       ident , … , ident at level num [binderinterp]
                       ident , … , ident at next level [binderinterp]
                       ident explicit_subentry
                       left associativity
                       right associativity
                       no associativity
                       only parsing
                       only printing
                       format string
explicit_subentry ::=  ident
                       global
                       bigint
                       [strict] pattern [at level num]
                       binder
                       closed binder
                       constr [binderinterp]
                       constr at level num [binderinterp]
                       constr at next level [binderinterp]
                       custom [binderinterp]
                       custom at level num [binderinterp]
                       custom at next level [binderinterp]
binderinterp      ::=  as ident
                       as pattern
                       as strict pattern
```

Note: No typing of the denoted expression is performed at definition time. Type checking is done only at the time of use of the notation.

Note: Some examples of Notation may be found in the files composing the initial state of Coq (see directory

`$COQLIB/theories/Init`).

Note: The notation `"{ x }"` has a special status in the main grammars of terms and patterns so 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"`).

Note: Notations such as `"( p | q )"` (or starting with `"( x | "`, more generally) are deprecated as they conflict with the syntax for nested disjunctive patterns (see *Extended pattern matching*), and are not honored in pattern expressions.

Warning: **`Use of`** ***`string`*** **`Notation is deprecated as it is inconsistent with pattern syntax.`**
This warning is disabled by default to avoid spurious diagnostics due to legacy notation in the Coq standard library. It can be turned on with the `-w disj-pattern-notation` flag.

Persistence of notations

Notations disappear when a section is closed.

Command: `Local Notation` *`notation`*
Notations survive modules unless the command `Local Notation` is used instead of *`Notation`*.

Command: `Local Declare Custom Entry` *`ident`*
Custom entries survive modules unless the command `Local Declare Custom Entry` is used instead of *`Declare Custom Entry`*.

6.1.2 Interpretation scopes

An *interpretation scope* is a set of notations for terms with their interpretations. Interpretation scopes provide a weak, purely syntactical form of notation overloading: the same notation, for instance the infix symbol `+`, can be used to denote distinct definitions of the additive operator. Depending on which interpretation scopes are currently open, the interpretation is different. Interpretation scopes can include an interpretation for numerals and strings, either at the OCaml level or using *`Numeral Notation`* or *`String Notation`*.

Command: `Declare Scope` *`scope`*
This adds a new scope named *`scope`*. Note that the initial state of Coq declares by default the following interpretation scopes: `core_scope`, `type_scope`, `function_scope`, `nat_scope`, `bool_scope`, `list_scope`, `int_scope`, `uint_scope`.

The syntax to associate a notation to a scope is given *above*. Here is a typical example which declares the notation for conjunction in the scope `type_scope`.

```
Notation "A /\ B" := (and A B) : type_scope.
```

Note: A notation not defined in a scope is called a *lonely* notation. No example of lonely notations can be found in the initial state of Coq though.

Global interpretation rules for notations

At any time, the interpretation of a notation for a 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 opened) 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).

Command: `Open Scope` *`scope`*

The command to add a scope to the interpretation scope stack is `Open Scope` *`scope`*.

Command: `Close 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 its invocations.

Note: `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.

Command: `Local Open Scope` *`scope`*`.`
Command: `Local Close Scope` *`scope`*`.`

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

Command: `Global Open Scope` *`scope`*`.`
Command: `Global Close Scope` *`scope`*`.`

These variants survive sections. They behave as if Global were absent when not inside a section.

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`*`)%`*`ident`* (or simply *`term`*`%`*`ident`* for atomic terms), where *`ident`* 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 *`ident`*.

Command: `Delimit Scope` *`scope`* `with` *`ident`*

To bind a delimiting key to a scope, use the command `Delimit Scope` *`scope`* `with` *`ident`*

Command: `Undelimit Scope` *scope*

To remove a delimiting key of a scope, use the command `Undelimit Scope` *scope*

Binding arguments of a constant to an interpretation scope

Command: `Arguments` *qualid* [*name*%*ident*]+

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` *qualid* [*name*%*ident*]+ where the list is a prefix of the arguments of `qualid` optionally annotated with their scope *ident*. Grouping round parentheses can be used to decorate multiple arguments with the same scope. *ident* can be either a scope name or its delimiting key. For example the following command puts the first two arguments of `plus_fct` in the scope delimited by the key `F` (`Rfun_scope`) and the last argument in the scope delimited by the key `R` (`R_scope`).

```
Arguments plus_fct (f1 f2)%F x%R.
```

The `Arguments` command accepts scopes decoration to all grouping parentheses. In the following example arguments A and B are marked as maximally inserted implicit arguments and are put into the `type_scope` scope.

```
Arguments respectful {A B}%type (R R')%signature _ _.
```

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 scope bound (if any) to this argument. The effect of the scope is limited to the argument itself. It does not propagate to subterms but the subterms that, after interpretation of the notation, turn to be themselves arguments of a reference are interpreted accordingly to the argument scopes bound to this reference.

Variant: `Arguments` *qualid* `: clear scopes`

This command can be used to clear argument scopes of *qualid*.

Variant: `Arguments` *qualid* [*name*%*ident*]+ `: extra scopes`

Defines extra argument scopes, to be used in case of coercion to `Funclass` (see the *Implicit Coercions* chapter) or with a computed type.

Variant: `Global Arguments` *qualid* [*name*%*ident*]+

This behaves like `Arguments qualid` [*name*%*ident*]+ but survives when a section is closed instead of stopping working at section closing. Without the `Global` modifier, the effect of the command stops when the section it belongs to ends.

Variant: `Local Arguments` *qualid* [*name*%*ident*]+

This behaves like `Arguments` *qualid* [*name*%*ident*]+ but does not survive modules and files. Without the `Local` modifier, the effect of the command is visible from within other modules or files.

See also:

The command *About* can be used to show the scopes bound to the arguments of a function.

Note: In notations, the subterms matching the identifiers of the notations are interpreted in the scope in which the identifiers occurred at the time of the declaration of the notation. Here is an example:

```
Parameter g : bool -> bool.
    g is declared

Declare Scope mybool_scope.
Notation "@@" := true (only parsing) : bool_scope.
    Setting notation at level 0.

Notation "@@" := false (only parsing): mybool_scope.
Bind Scope bool_scope with bool.
Notation "# x #" := (g x) (at level 40).
Check # @@ #.
    # true #
         : bool

Arguments g _%mybool_scope.
Check # @@ #.
    # true #
         : bool

Delimit Scope mybool_scope with mybool.
Check # @@%mybool #.
    # false #
         : bool
```

Binding types of arguments to an interpretation scope

Command: `Bind Scope` *`scope`* `with` *`qualid`*

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. When a scope *scope* is bound to a type *type*, any function gets its arguments of type *type* interpreted by default in scope *scope* (this default behavior can however be overwritten by explicitly using the command *Arguments*).

Whether the argument of a function has some type `type` is determined statically. For instance, if `f` is a polymorphic function of type `forall X:Type, X -> X` and type `t` is bound to a scope `scope`, then `a` of type `t` in `f t a` is not recognized as an argument to be interpreted in scope `scope`.

More generally, any coercion *class* (see the *Implicit Coercions* chapter) can be bound to an interpretation scope. The command to do it is `Bind Scope` *`scope`* `with` *`class`*

```
Parameter U : Set.
Declare Scope U_scope.
Bind Scope U_scope with U.
Parameter Uplus : U -> U -> U.
Parameter P : forall T:Set, T -> U -> Prop.
Parameter f : forall T:Set, T -> U.
Infix "+" := Uplus : U_scope.
Unset Printing Notations.
Open Scope nat_scope.

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 (Nat.add x t) (Uplus (f nat (Nat.add y1 y2)) z)
         : forall (_ : nat) (_ : nat) (_ : nat) (_ : U) (_ : nat), Prop
```

Note: When active, a bound scope has effect on all defined functions (even if they are defined after the *Bind Scope* directive), except if argument scopes were assigned explicitly using the *Arguments* command.

Note: The scopes `type_scope` and `function_scope` also have a local effect on interpretation. See the next section.

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. It is delimited by the key `type`, and bound to the coercion class `Sortclass`. It is also used in certain situations where an expression is statically known to be a type, including the conclusion and the type of hypotheses within an Ltac goal match (see *Pattern matching on goals*), the statement of a theorem, the type of a definition, the type of a binder, the domain and codomain of implication, the codomain of products, and more generally any type argument of a declared or defined constant.

The `function_scope` interpretation scope

The scope `function_scope` also has a special status. It is temporarily activated each time the argument of a global reference is recognized to be a `Funclass` instance, i.e., of type `forall x:A, B` or `A -> B`.

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 Scope*.

`type_scope` This scope includes infix * for product types and infix + for sum types. It is delimited by the key `type`, and bound to the coercion class `Sortclass`, as described above.

`function_scope` This scope is delimited by the key `function`, and bound to the coercion class `Funclass`, as described above.

`nat_scope` This scope includes the standard arithmetical operators and relations on type nat. Positive integer numerals in this scope are mapped to their canonical representent built from `O` and `S`. The scope is delimited by the key `nat`, and bound to the type `nat` (see above).

`N_scope` This scope includes the standard arithmetical operators and relations on type `N` (binary natural numbers). It is delimited by the key `N` and comes with an interpretation for numerals as closed terms of type `N`.

`Z_scope` This scope includes the standard arithmetical operators and relations on type `Z` (binary integer numbers). It is delimited by the key `Z` and comes with an interpretation for numerals as closed terms of type `Z`.

`positive_scope` This scope 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 terms of type `positive`.

`Q_scope` This scope includes the standard arithmetical operators and relations on type `Q` (rational numbers defined as fractions of an integer and a strictly positive integer modulo the equality of the numerator-denominator cross-product) and comes with an interpretation for numerals as closed terms of type `Q`.

`Qc_scope` This scope includes the standard arithmetical operators and relations on the type `Qc` of rational numbers defined as the type of irreducible fractions of an integer and a strictly positive integer.

`R_scope` This scope includes the standard arithmetical operators and relations on type `R` (axiomatic real numbers). It is delimited by the key `R` and comes with an interpretation for numerals using the `IZR` morphism from binary integer numbers to `R` and `Z.pow_pos` for potential exponent parts.

`bool_scope` This scope includes notations for the boolean operators. It is delimited by the key `bool`, and bound to the type `bool` (see above).

`list_scope` This scope includes notations for the list operators. It is delimited by the key `list`, and bound to the type `list` (see above).

`core_scope` This scope includes the notation for pairs. It is delimited by the key `core`.

`string_scope` This scope includes notation for strings as elements of the type string. Special characters and escaping follow Coq conventions on strings (see *Lexical conventions*). Especially, there is no convention to visualize non printable characters of a string. The file `String.v` shows an example that contains quotes, a newline and a beep (i.e. the ASCII character of code 7).

`char_scope` This scope includes interpretation for all strings of the form `"c"` where `c` is an ASCII character, or of the form `"nnn"` where nnn is a three-digits number (possibly with leading 0's), or of the form `""""`. Their respective denotations are the ASCII code of `c`, the decimal ASCII code `nnn`, or the ascii code of the character `"` (i.e. the ASCII code 34), all of them being represented in the type `ascii`.

Displaying information about scopes

Command: `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 opened 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 *`scope`* is interpreted.

Command: `Print Scopes`

This displays all the notations, delimiting keys and corresponding classes of all the existing interpretation scopes. It also displays the lonely notations.

Variant: `Print Scope` *`scope`*

This displays all the notations defined in the interpretation scope *`scope`*. It also displays the delimiting key if any and the class to which the scope is bound, if any.

6.1.3 Abbreviations

Command: `[Local]`? `Notation` *`ident`* `[`*`ident`*`]`+ `:=` *`term`* `[(only parsing)]`? `.`

An *abbreviation* is a name, possibly applied to arguments, that denotes a (presumably) more complex expression. Here are examples:

```
Notation Nlist := (list nat).
```

```
Check 1 :: 2 :: 3 :: nil.
    1 :: 2 :: 3 :: nil
         : Nlist

Notation reflexive R := (forall x, R x x).

Check forall A:Prop, A <-> A.
    reflexive iff
         : Prop

Check reflexive iff.
    reflexive iff
         : Prop
```

An abbreviation expects no precedence nor associativity, since it is parsed as an usual application. Abbreviations are used as much as possible by the Coq printers unless the modifier `(only parsing)` is given.

An abbreviation is bound to an absolute name as an ordinary definition is and it also can be referred to by a qualified name.

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 they are used. Especially, abbreviations can be bound to terms with holes (i.e. with “_”). For example:

```
Definition explicit_id (A:Set) (a:A) := a.

Notation id := (explicit_id _).

Check (id 0).
    id 0
         : nat
```

Abbreviations disappear when a section is closed. No typing of the denoted expression is performed at definition time. Type checking is done only at the time of use of the abbreviation.

6.1.4 Numeral notations

Command: `Numeral Notation` ident_1 ident_2 ident_3 `:` *scope*.

This command allows the user to customize the way numeral literals are parsed and printed.

The token ident_1 should be the name of an inductive type, while ident_2 and ident_3 should be the names of the parsing and printing functions, respectively. The parsing function ident_2 should have one of the following types:

- `Decimal.int ->` ident_1
- `Decimal.int -> option` ident_1
- `Decimal.uint ->` ident_1
- `Decimal.uint -> option` ident_1
- `Z ->` ident_1
- `Z -> option` ident_1
- `Decimal.decimal ->` ident_1

- `Decimal.decimal -> option` *ident*$_1$

And the printing function *ident*$_3$ should have one of the following types:

- *ident*$_1$ `-> Decimal.int`
- *ident*$_1$ `-> option Decimal.int`
- *ident*$_1$ `-> Decimal.uint`
- *ident*$_1$ `-> option Decimal.uint`
- *ident*$_1$ `-> Z`
- *ident*$_1$ `-> option Z`
- *ident*$_1$ `-> Decimal.decimal`
- *ident*$_1$ `-> option Decimal.decimal`

When parsing, the application of the parsing function *ident*$_2$ to the number will be fully reduced, and universes of the resulting term will be refreshed.

Note that only fully-reduced ground terms (terms containing only function application, constructors, inductive type families, sorts, and primitive integers) will be considered for printing.

Variant: `Numeral Notation` *ident*$_1$ *ident*$_2$ *ident*$_3$ `:` *scope* `(warning after` *num*`).`
When a literal larger than *num* is parsed, a warning message about possible stack overflow, resulting from evaluating *ident*$_2$, will be displayed.

Variant: `Numeral Notation` *ident*$_1$ *ident*$_2$ *ident*$_3$ `:` *scope* `(abstract after` *num*`).`
When a literal `m` larger than *num* is parsed, the result will be `(`*ident*$_2$ `m)`, without reduction of this application to a normal form. Here `m` will be a `Decimal.int` or `Decimal.uint` or `Z`, depending on the type of the parsing function *ident*$_2$. This allows for a more compact representation of literals in types such as `nat`, and limits parse failures due to stack overflow. Note that a warning will be emitted when an integer larger than *num* is parsed. Note that `(abstract after` *num*`)` has no effect when *ident*$_2$ lands in an `option` type.

Error: `Cannot interpret this number as a value of type` *type*
The numeral notation registered for *type* does not support the given numeral. This error is given when the interpretation function returns `None`, or if the interpretation is registered only for integers or non-negative integers, and the given numeral has a fractional or exponent part or is negative.

Error: *ident* `should go from Decimal.int to` *type* `or (option` *type*`). Instead of Decimal.int, the types`
The parsing function given to the *Numeral Notation* vernacular is not of the right type.

Error: *ident* `should go from` *type* `to Decimal.int or (option Decimal.int). Instead of Decimal.int, th`
The printing function given to the *Numeral Notation* vernacular is not of the right type.

Error: *type* `is not an inductive type.`
Numeral notations can only be declared for inductive types with no arguments.

Error: `Unexpected term` *term* `while parsing a numeral notation.`
Parsing functions must always return ground terms, made up of applications of constructors, inductive types, and primitive integers. Parsing functions may not return terms containing axioms, bare (co)fixpoints, lambdas, etc.

Error: `Unexpected non-option term` *term* `while parsing a numeral notation.`
Parsing functions expected to return an `option` must always return a concrete `Some` or `None` when applied to a concrete numeral expressed as a decimal. They may not return opaque constants.

Error: `Cannot interpret in` *scope* `because` *ident* `could not be found in the current environment.`

The inductive type used to register the numeral notation is no longer available in the environment. Most likely, this is because the numeral notation was declared inside a functor for an inductive type inside the functor. This use case is not currently supported.

Alternatively, you might be trying to use a primitive token notation from a plugin which forgot to specify which module you must `Require` for access to that notation.

Error: `Syntax error: [prim:reference] expected after 'Notation' (in [vernac:command]).`

The type passed to *Numeral Notation* must be a single identifier.

Error: `Syntax error: [prim:reference] expected after [prim:reference] (in [vernac:command]).`

Both functions passed to *Numeral Notation* must be single identifiers.

Error: `The reference` *ident* `was not found in the current environment.`

Identifiers passed to *Numeral Notation* must exist in the global environment.

Error: *ident* `is bound to a notation that does not denote a reference.`

Identifiers passed to *Numeral Notation* must be global references, or notations which denote to single identifiers.

Warning: `Stack overflow or segmentation fault happens when working with large numbers in` *type* `(thr`

When a *Numeral Notation* is registered in the current scope with `(warning after` *num*`)`, this warning is emitted when parsing a numeral greater than or equal to *num*.

Warning: `To avoid stack overflow, large numbers in` *type* `are interpreted as applications of` *ident*$_2$`.`

When a *Numeral Notation* is registered in the current scope with `(abstract after` *num*`)`, this warning is emitted when parsing a numeral greater than or equal to *num*. Typically, this indicates that the fully computed representation of numerals can be so large that non-tail-recursive OCaml functions run out of stack space when trying to walk them.

For example

```
Check 90000.
    Toplevel input, characters 0-12:
    > Check 90000.
    > ^^^^^^^^^^^^
    Warning: To avoid stack overflow, large numbers in nat are interpreted as
    applications of Nat.of_uint. [abstract-large-number,numbers]
    Nat.of_uint 90000
         : nat
```

Warning: `The 'abstract after' directive has no effect when the parsing function (`*ident*$_2$`) targets a`

As noted above, the `(abstract after` *num*`)` directive has no effect when *ident*$_2$ lands in an `option` type.

6.1.5 String notations

Command: `String Notation` *ident*$_1$ *ident*$_2$ *ident*$_3$ `:` *scope*`.`

This command allows the user to customize the way strings are parsed and printed.

The token *ident*$_1$ should be the name of an inductive type, while *ident*$_2$ and *ident*$_3$ should be the names of the parsing and printing functions, respectively. The parsing function *ident*$_2$ should have one of the following types:

- `Byte.byte ->` *ident*$_1$
- `Byte.byte -> option` *ident*$_1$
- `list Byte.byte ->` *ident*$_1$

- `list Byte.byte -> option` *ident*$_1$

And the printing function *ident*$_3$ should have one of the following types:

- *ident*$_1$ `-> Byte.byte`
- *ident*$_1$ `-> option Byte.byte`
- *ident*$_1$ `-> list Byte.byte`
- *ident*$_1$ `-> option (list Byte.byte)`

When parsing, the application of the parsing function *ident*$_2$ to the string will be fully reduced, and universes of the resulting term will be refreshed.

Note that only fully-reduced ground terms (terms containing only function application, constructors, inductive type families, sorts, and primitive integers) will be considered for printing.

Error: Cannot interpret this string as a value of type *type*

The string notation registered for *type* does not support the given string. This error is given when the interpretation function returns `None`.

Error: *ident* should go from Byte.byte or (list Byte.byte) to *type* or (option *type*).

The parsing function given to the *String Notation* vernacular is not of the right type.

Error: *ident* should go from *type* to Byte.byte or (option Byte.byte) or (list Byte.byte) or (option

The printing function given to the *String Notation* vernacular is not of the right type.

Error: *type* is not an inductive type.

String notations can only be declared for inductive types with no arguments.

Error: Unexpected term *term* while parsing a string notation.

Parsing functions must always return ground terms, made up of applications of constructors, inductive types, and primitive integers. Parsing functions may not return terms containing axioms, bare (co)fixpoints, lambdas, etc.

Error: Unexpected non-option term *term* while parsing a string notation.

Parsing functions expected to return an `option` must always return a concrete `Some` or `None` when applied to a concrete string expressed as a decimal. They may not return opaque constants.

Error: Cannot interpret in *scope* because *ident* could not be found in the current environment.

The inductive type used to register the string notation is no longer available in the environment. Most likely, this is because the string notation was declared inside a functor for an inductive type inside the functor. This use case is not currently supported.

Alternatively, you might be trying to use a primitive token notation from a plugin which forgot to specify which module you must `Require` for access to that notation.

Error: Syntax error: [prim:reference] expected after 'Notation' (in [vernac:command]).

The type passed to *String Notation* must be a single identifier.

Error: Syntax error: [prim:reference] expected after [prim:reference] (in [vernac:command]).

Both functions passed to *String Notation* must be single identifiers.

Error: The reference *ident* was not found in the current environment.

Identifiers passed to *String Notation* must exist in the global environment.

Error: *ident* is bound to a notation that does not denote a reference.

Identifiers passed to *String Notation* must be global references, or notations which denote to single identifiers.

6.1.6 Tactic Notations

Tactic notations allow to customize the syntax of tactics. They have the following syntax:

```
tacn                  ::=  Tactic Notation [tactic_level] [prod_item … prod_item] := tactic.
prod_item             ::=  string | tactic_argument_type(ident)
tactic_level          ::=  (at level num)
tactic_argument_type  ::=  ident | simple_intropattern | reference
                           hyp | hyp_list | ne_hyp_list
                           constr | uconstr | constr_list | ne_constr_list
                           integer | integer_list | ne_integer_list
                           int_or_var | int_or_var_list | ne_int_or_var_list
                           tactic | tactic0 | tactic1 | tactic2 | tactic3
                           tactic4 | tactic5
```

Command: `Tactic Notation` `(at level` *`num`*`)`? *`prod_item`*+ `:=` *`tactic`*`.`

A tactic notation extends the parser and pretty-printer of tactics with a new rule made of the list of production items. It then evaluates into the tactic expression `tactic`. For simple tactics, it is recommended to use a terminal symbol, i.e. a string, for the first production item. The tactic level indicates the parsing precedence of the tactic notation. This information is particularly relevant for notations of tacticals. Levels 0 to 5 are available (default is 5).

Command: `Print Grammar tactic`

To know the parsing precedences of the existing tacticals, use the command `Print Grammar tactic`.

Each type of tactic argument has a specific semantic regarding how it is parsed and how it is interpreted. The semantic is described in the following table. The last command gives examples of tactics which use the corresponding kind of argument.

Tactic argument type	parsed as	interpreted as	as in tactic
`ident`	identifier	a user-given name	intro
`simple_intropattern`	simple_intropattern	an introduction pattern	assert as
`hyp`	identifier	a hypothesis defined in context	clear
`reference`	qualified identifier	a global reference of term	unfold
`constr`	term	a term	exact
`uconstr`	term	an untyped term	refine
`integer`	integer	an integer	
`int_or_var`	identifier or integer	an integer	do
`tactic`	tactic at level 5	a tactic	
`tacticn`	tactic at level n	a tactic	
entry`_list`	list of *entry*	a list of how *entry* is interpreted	
`ne_`*entry*`_list`	non-empty list of *entry*	a list of how *entry* is interpreted	

Note: In order to be bound in tactic definitions, each syntactic entry for argument type must include the case of a simple $\mathrm{L}_{\mathrm{tac}}$ identifier as part of what it parses. This is naturally the case for `ident`, `simple_intropattern`, `reference`, `constr`, ... but not for `integer`. This is the reason for introducing a special entry `int_or_var` which evaluates to integers only but which syntactically includes identifiers in order to be usable in tactic definitions.

Note: The *entry*`_list` and `ne_`*entry*`_list` entries can be used in primitive tactics or in other notations at places where a list of the underlying entry can be used: entry is either `constr`, `hyp`, `integer` or `int_or_var`.

Variant: `Local Tactic Notation`

Tactic notations disappear when a section is closed. They survive when a module is closed unless the command `Local Tactic Notation` is used instead of *`Tactic Notation`*.

6.2 Proof schemes

6.2.1 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:

Command: `Scheme` *$ident_1$* `:=` `Induction for` *$ident_2$* `Sort` *sort* `with` *$ident_i$* `:=` `Induction for` *$ident_j$* `Sort` *sor*

This command is a high-level tool for generating automatically (possibly mutual) induction principles for given types and sorts. Each *$ident_j$* is a different inductive type identifier belonging to the same package of mutual inductive definitions. The command generates the *$ident_i$*s to be mutually recursive definitions. Each term *$ident_i$* proves a general principle of mutual induction for objects in type *$ident_j$*.

Variant: `Scheme` *ident* `:=` `Minimality for` *ident* `Sort` *sort* `with` *ident* `:=` `Minimality for` *ident*`'` `Sort` *sort* *

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

Variant: `Scheme Equality for` *ident*

Tries to generate a Boolean equality and a proof of the decidability of the usual equality. If `ident` involves some other inductive types, their equality has to be defined first.

Variant: `Scheme Induction for` *ident* `Sort` *sort* `with Induction for` *ident* `Sort` *sort* *

If you do not provide the name of the schemes, they will be automatically computed from the sorts involved (works also with Minimality).

Example

Induction scheme for tree and forest.

A mutual induction principle for tree and forest in sort `Set` can be defined using the command

```
Inductive tree : Set := node : A -> forest -> tree
with forest : Set :=
    leaf : B -> forest
  | cons : tree -> forest -> forest.
    tree, forest are defined
    tree_rect is defined
    tree_ind is defined
    tree_rec is defined
    tree_sind is defined
    forest_rect is defined
    forest_ind is defined
    forest_rec is defined
    forest_sind is defined
```

(continues on next page)

(continued from previous page)

```
Scheme tree_forest_rec := Induction for tree Sort Set
  with forest_tree_rec := Induction for forest Sort Set.
    forest_tree_rec is defined
    tree_forest_rec is defined
    tree_forest_rec, forest_tree_rec are recursively defined
```

You may now look at the type of tree_forest_rec:

```
Check tree_forest_rec.
    tree_forest_rec
         : forall (P : tree -> Set) (P0 : forest -> Set),
           (forall (a : A) (f : forest), P0 f -> P (node a f)) ->
           (forall b : B, P0 (leaf b)) ->
           (forall t : tree, P t -> forall f1 : forest, P0 f1 -> P0 (cons t f1)) ->
           forall t : tree, P t
```

This principle involves two different predicates for trees andforests; it also has three premises each one corresponding to a constructor of one of the inductive definitions.

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

Example

Predicates odd and even on naturals.

Let odd and even be inductively defined as:

```
Inductive odd : nat -> Prop := oddS : forall n:nat, even n -> odd (S n)
with even : nat -> Prop :=
  | evenO : even 0
  | evenS : forall n:nat, odd n -> even (S n).
    odd, even are defined
    odd_ind is defined
    odd_sind is defined
    even_ind is defined
    even_sind is defined
```

The following command generates a powerful elimination principle:

```
Scheme odd_even := Minimality for odd Sort Prop
with even_odd := Minimality for even Sort Prop.
    even_odd is defined
    odd_even is defined
    odd_even, even_odd are recursively defined
```

The type of odd_even for instance will be:

```
Check odd_even.
    odd_even
         : forall P P0 : nat -> Prop,
           (forall n : nat, even n -> P0 n -> P (S n)) ->
           P0 0 ->
           (forall n : nat, odd n -> P n -> P0 (S n)) ->
           forall n : nat, odd n -> P n
```

The type of `even_odd` shares the same premises but the conclusion is `(n:nat)(even n)->(P0 n)`.

Automatic declaration of schemes

Flag: **`Elimination Schemes`**

Enables automatic declaration of induction principles when defining a new inductive type. Defaults to on.

Flag: **`Nonrecursive Elimination Schemes`**

Enables automatic declaration of induction principles for types declared with the *`Variant`* and *`Record`* commands. Defaults to off.

Flag: **`Case Analysis Schemes`**

This flag governs the generation of case analysis lemmas for inductive types, i.e. corresponding to the pattern matching term alone and without fixpoint.

Flag: **`Boolean Equality Schemes`**

Flag: **`Decidable Equality Schemes`**

These flags control the automatic declaration of those Boolean equalities (see the second variant of `Scheme`).

> **Warning:** You have to be careful with these flags since Coq may now reject well-defined inductive types because it cannot compute a Boolean equality for them.

Flag: **`Rewriting Schemes`**

This flag governs generation of equality-related schemes such as congruence.

Combined Scheme

Command: **`Combined Scheme`** *`ident`* **`from`** *`ident`*$_i$ + ,

This command is a tool for combining induction principles generated by the *`Scheme`* command. Each *`ident`*$_i$ is a different inductive principle that must belong to the same package of mutual inductive principle definitions. This command generates *`ident`* to be the conjunction of the principles: it is built from the common premises of the principles and concluded by the conjunction of their conclusions. In the case where all the inductive principles used are in sort `Prop`, the propositional conjunction `and` is used, otherwise the simple product `prod` is used instead.

Example

We can define the induction principles for trees and forests using:

```
Scheme tree_forest_ind := Induction for tree Sort Prop
with forest_tree_ind := Induction for forest Sort Prop.
    forest_tree_ind is defined
    tree_forest_ind is defined
    tree_forest_ind, forest_tree_ind are recursively defined
```

Then we can build the combined induction principle which gives the conjunction of the conclusions of each individual principle:

```
Combined Scheme tree_forest_mutind from tree_forest_ind,forest_tree_ind.
    tree_forest_mutind is defined
    tree_forest_mutind is recursively defined
```

The type of tree_forest_mutind will be:

```
Check tree_forest_mutind.
    tree_forest_mutind
         : forall (P : tree -> Prop) (P0 : forest -> Prop),
           (forall (a : A) (f : forest), P0 f -> P (node a f)) ->
           (forall b : B, P0 (leaf b)) ->
           (forall t : tree, P t -> forall f1 : forest, P0 f1 -> P0 (cons t f1)) ->
           (forall t : tree, P t) /\ (forall f2 : forest, P0 f2)
```

Example

We can also combine schemes at sort `Type`:

```
Scheme tree_forest_rect := Induction for tree Sort Type
with forest_tree_rect := Induction for forest Sort Type.
    forest_tree_rect is defined
    tree_forest_rect is defined
    tree_forest_rect, forest_tree_rect are recursively defined
```

```
Combined Scheme tree_forest_mutrect from tree_forest_rect, forest_tree_rect.
    tree_forest_mutrect is defined
    tree_forest_mutrect is recursively defined
```

```
Check tree_forest_mutrect.
    tree_forest_mutrect
         : forall (P : tree -> Type) (P0 : forest -> Type),
           (forall (a : A) (f : forest), P0 f -> P (node a f)) ->
           (forall b : B, P0 (leaf b)) ->
           (forall t : tree, P t -> forall f1 : forest, P0 f1 -> P0 (cons t f1)) ->
           (forall t : tree, P t) * (forall f2 : forest, P0 f2)
```

6.2.2 Generation of induction principles with `Functional Scheme`

Command: `Functional Scheme` *ident*$_0$ `:= Induction for` *ident*`'` `Sort` *sort* `with` *ident*$_i$ `:= Induction for` *iden*

This command is a high-level experimental tool for generating automatically induction principles corresponding to (possibly mutually recursive) functions. First, it must be made available via `Require Import FunInd`. Each *ident*$_i$ is a different mutually defined function name (the names must be in the same order as when they were defined). This command generates the induction principle for each *ident*$_i$, following the recursive structure and case analyses of the corresponding function *ident*$_i$`'`.

Warning: There is a difference between induction schemes generated by the command *`Functional Scheme`* and these generated by the *`Function`*. Indeed, *`Function`* generally produces smaller principles that are closer to how a user would implement them. See *Advanced recursive functions* for details.

Example

Induction scheme for div2.

We define the function div2 as follows:

```
Require Import FunInd.
    [Loading ML file extraction_plugin.cmxs ... done]
    [Loading ML file recdef_plugin.cmxs ... done]

Require Import Arith.
    [Loading ML file newring_plugin.cmxs ... done]

Fixpoint div2 (n:nat) : nat :=
match n with
| O => 0
| S O => 0
| S (S n') => S (div2 n')
end.
    div2 is defined
    div2 is recursively defined (decreasing on 1st argument)
```

The definition of a principle of induction corresponding to the recursive structure of `div2` is defined by the command:

```
Functional Scheme div2_ind := Induction for div2 Sort Prop.
    div2_equation is defined
    div2_ind is defined
```

You may now look at the type of div2_ind:

```
Check div2_ind.
    div2_ind
         : forall P : nat -> nat -> Prop,
           (forall n : nat, n = 0 -> P 0 0) ->
           (forall n n0 : nat, n = S n0 -> n0 = 0 -> P 1 0) ->
           (forall n n0 : nat,
            n = S n0 ->
            forall n' : nat,
            n0 = S n' -> P n' (div2 n') -> P (S (S n')) (S (div2 n'))) ->
           forall n : nat, P n (div2 n)
```

We can now prove the following lemma using this principle:

```
Lemma div2_le' : forall n:nat, div2 n <= n.
    1 subgoal

      ============================
      forall n : nat, div2 n <= n

intro n.
    1 subgoal

      n : nat
      ============================
      div2 n <= n

pattern n, (div2 n).
```

(continues on next page)

(continued from previous page)

```
    1 subgoal

      n : nat
      ============================
      (fun n0 n1 : nat => n1 <= n0) n (div2 n)

apply div2_ind; intros.
    3 subgoals

      n, n0 : nat
      e : n0 = 0
      ============================
      0 <= 0

    subgoal 2 is:
     0 <= 1
    subgoal 3 is:
     S (div2 n') <= S (S n')

auto with arith.
    2 subgoals

      n, n0, n1 : nat
      e : n0 = S n1
      e0 : n1 = 0
      ============================
      0 <= 1

    subgoal 2 is:
     S (div2 n') <= S (S n')

auto with arith.
    1 subgoal

      n, n0, n1 : nat
      e : n0 = S n1
      n' : nat
      e0 : n1 = S n'
      H : div2 n' <= n'
      ============================
      S (div2 n') <= S (S n')

simpl; auto with arith.
    No more subgoals.

Qed.
```

We can use directly the functional induction (*function induction*) tactic instead of the pattern/apply trick:

```
Reset div2_le'.
Lemma div2_le : forall n:nat, div2 n <= n.
    1 subgoal

      ============================
      forall n : nat, div2 n <= n
```

(continues on next page)

(continued from previous page)

```
intro n.
    1 subgoal

      n : nat
      ============================
      div2 n <= n

functional induction (div2 n).
    3 subgoals

      ============================
      0 <= 0

    subgoal 2 is:
     0 <= 1
    subgoal 3 is:
     S (div2 n') <= S (S n')

auto with arith.
    2 subgoals

      ============================
      0 <= 1

    subgoal 2 is:
     S (div2 n') <= S (S n')

auto with arith.
    1 subgoal

      n' : nat
      IHn0 : div2 n' <= n'
      ============================
      S (div2 n') <= S (S n')

auto with arith.
    No more subgoals.

Qed.
```

Example

Induction scheme for tree_size.

We define trees by the following mutual inductive type:

```
Axiom A : Set.
    A is declared

Inductive tree : Set :=
node : A -> forest -> tree
with forest : Set :=
| empty : forest
| cons : tree -> forest -> forest.
```

(continues on next page)

(continued from previous page)

```
    tree, forest are defined
    tree_rect is defined
    tree_ind is defined
    tree_rec is defined
    tree_sind is defined
    forest_rect is defined
    forest_ind is defined
    forest_rec is defined
    forest_sind is defined
```

We define the function tree_size that computes the size of a tree or a forest. Note that we use `Function` which generally produces better principles.

```
Require Import FunInd.
Function tree_size (t:tree) : nat :=
match t with
| node A f => S (forest_size f)
end
with forest_size (f:forest) : nat :=
match f with
| empty => 0
| cons t f' => (tree_size t + forest_size f')
end.
    tree_size is defined
    forest_size is defined
    tree_size, forest_size are recursively defined
    (decreasing respectively on 1st, 1st arguments)
    tree_size_equation is defined
    tree_size_rect is defined
    tree_size_ind is defined
    tree_size_rec is defined
    forest_size_equation is defined
    forest_size_rect is defined
    forest_size_ind is defined
    forest_size_rec is defined
    R_tree_size_correct is defined
    R_forest_size_correct is defined
    R_tree_size_complete is defined
    R_forest_size_complete is defined
```

Notice that the induction principles `tree_size_ind` and `forest_size_ind` generated by `Function` are not mutual.

```
Check tree_size_ind.
    tree_size_ind
         : forall P : tree -> nat -> Prop,
           (forall (t : tree) (A : A) (f : forest),
            t = node A f -> P (node A f) (S (forest_size f))) ->
           forall t : tree, P t (tree_size t)
```

Mutual induction principles following the recursive structure of `tree_size` and `forest_size` can be generated by the following command:

```
Functional Scheme tree_size_ind2 := Induction for tree_size Sort Prop
with forest_size_ind2 := Induction for forest_size Sort Prop.
    tree_size_ind2 is defined
    forest_size_ind2 is defined
```

You may now look at the type of `tree_size_ind2`:

```
Check tree_size_ind2.
    tree_size_ind2
         : forall (P : tree -> nat -> Prop) (P0 : forest -> nat -> Prop),
           (forall (t : tree) (A : A) (f : forest),
            t = node A f ->
            P0 f (forest_size f) -> P (node A f) (S (forest_size f))) ->
           (forall f0 : forest, f0 = empty -> P0 empty 0) ->
           (forall (f1 : forest) (t : tree) (f' : forest),
            f1 = cons t f' ->
            P t (tree_size t) ->
            P0 f' (forest_size f') ->
            P0 (cons t f') (tree_size t + forest_size f')) ->
           forall t : tree, P t (tree_size t)
```

6.2.3 Generation of inversion principles with `Derive Inversion`

Command: `Derive Inversion` *`ident`* `with` *`ident`* `Sort` *`sort`*
Command: `Derive Inversion` *`ident`* `with (forall` *`binders`*`,` *`ident term`*`) Sort` *`sort`*

This command generates an inversion principle for the *`inversion ... using ...`* tactic. The first *`ident`* is the name of the generated principle. The second *`ident`* should be an inductive predicate, and *`binders`* the variables occurring in the term *`term`*. This command generates the inversion lemma for the sort *`sort`* corresponding to the instance `forall` *`binders`*, *`ident term`*. When applied, it is equivalent to having inverted the instance with the tactic `inversion`.

Variant: `Derive Inversion_clear` *`ident`* `with` *`ident`* `Sort` *`sort`*
Variant: `Derive Inversion_clear` *`ident`* `with (forall` *`binders`*`,` *`ident term`*`) Sort` *`sort`*

When applied, it is equivalent to having inverted the instance with the tactic inversion replaced by the tactic `inversion_clear`.

Variant: `Derive Dependent Inversion` *`ident`* `with` *`ident`* `Sort` *`sort`*
Variant: `Derive Dependent Inversion` *`ident`* `with (forall` *`binders`*`,` *`ident term`*`) Sort` *`sort`*

When applied, it is equivalent to having inverted the instance with the tactic `dependent inversion`.

Variant: `Derive Dependent Inversion_clear` *`ident`* `with` *`ident`* `Sort` *`sort`*
Variant: `Derive Dependent Inversion_clear` *`ident`* `with (forall` *`binders`*`,` *`ident term`*`) Sort` *`sort`*

When applied, it is equivalent to having inverted the instance with the tactic `dependent inversion_clear`.

Example

Consider the relation `Le` over natural numbers and the following parameter `P`:

```
Inductive Le : nat -> nat -> Set :=
| LeO : forall n:nat, Le 0 n
| LeS : forall n m:nat, Le n m -> Le (S n) (S m).
    Le is defined
    Le_rect is defined
    Le_ind is defined
    Le_rec is defined
    Le_sind is defined

Parameter P : nat -> nat -> Prop.
    P is declared
```

To generate the inversion lemma for the instance `(Le (S n) m)` and the sort `Prop`, we do:

```
Derive Inversion_clear leminv with (forall n m:nat, Le (S n) m) Sort Prop.
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:

```
Show.
    1 subgoal

      n, m : nat
      H : Le (S n) m
      ============================
      P n m

inversion H using leminv.
    1 subgoal

      n, m : nat
      H : Le (S n) m
      ============================
      forall m0 : nat, Le n m0 -> P n (S m0)
```

CHAPTER

SEVEN

PRACTICAL TOOLS

7.1 The Coq commands

There are three Coq commands:

- `coqtop`: the Coq toplevel (interactive mode);
- `coqc`: the Coq compiler (batch compilation);
- `coqchk`: the Coq checker (validation of compiled libraries).

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

7.1.1 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`.

There are two different binary images of Coq: the byte-code one and the native-code one (if OCaml provides a native-code compiler for your platform, which is supposed in the following). By default, `coqtop` executes the native-code version; run `coqtop.byte` to get the byte-code version.

The byte-code toplevel is based on an OCaml toplevel (to allow dynamic linking of tactics). You can switch to the OCaml toplevel with the command `Drop.`, and come back to the Coq toplevel with the command `Coqloop.loop();;`.

Flag: **`Coqtop Exit On Error`**

This flag, off by default, causes coqtop to exit with status code `1` if a command produces an error instead of recovering from it.

7.1.2 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 *Compiled files*).

Caution: The name *file* should be a regular Coq identifier as defined in Section *Lexical conventions*. It should contain only letters, digits or underscores (_). For example `/bar/foo/toto.v` is valid, but `/bar/foo/to-to.v` is not.

7.1.3 Customization at launch time

By resource file

When Coq is launched, with either `coqtop` or `coqc`, the resource file `$XDG_CONFIG_HOME/coq/coqrc.xxx`, if it exists, will be implicitly prepended to any document read by Coq, whether it is an interactive session or a file to compile. Here, `$XDG_CONFIG_HOME` is the configuration directory of the user (by default it's `~/.config`) and `xxx` is the version number (e.g. 8.8). If this file is not found, then the file `$XDG_CONFIG_HOME/coqrc` is searched. If not found, it is the file `~/.coqrc.xxx` which is searched, and, if still not found, the file `~/.coqrc`. If the latter is also absent, no resource file is loaded. You can also specify an arbitrary name for the resource file (see option `-init-file` below).

The resource 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`.

By environment variables

Load path can be specified to the Coq system by setting up `$COQPATH` environment variable. It is a list of directories separated by `:` (`;` on Windows). Coq will also honor `$XDG_DATA_HOME` and `$XDG_DATA_DIRS` (see Section *Libraries and filesystem*).

Some Coq commands call other Coq commands. In this case, they look for the commands in directory specified by `$COQBIN`. If this variable is not set, they look for the commands in the executable path.

The `$COQ_COLORS` environment variable can be used to specify the set of colors used by `coqtop` to highlight its output. It uses the same syntax as the `$LS_COLORS` variable from GNU's ls, that is, a colon-separated list of assignments of the form `name=`$\texttt{attr}^{*}_{;}$ where `name` is the name of the corresponding highlight tag and each `attr` is an ANSI escape code. The list of highlight tags can be retrieved with the `-list-tags` command-line option of `coqtop`.

The string uses ANSI escape codes to represent attributes. For example:

```
export COQ_COLORS="diff.added=4;48;2;0;0;240:diff.removed=41"
```

sets the highlights for added text in diffs to underlined (the 4) with a background RGB color (0, 0, 240) and for removed text in diffs to a red background. Note that if you specify `COQ_COLORS`, the predefined attributes are ignored.

By command line options

The following command-line options are recognized by the commands `coqc` and `coqtop`, unless stated otherwise:

-I *directory*, -include *directory* Add physical path *directory* to the OCaml loadpath.

> **See also:**
>
> *Names of libraries* and the command Declare ML Module Section *Compiled files*.

-Q *directory dirpath* Add physical path *directory* to the list of directories where Coq looks for a file and bind it to the logical directory *dirpath*. The subdirectory structure of *directory* is recursively available from Coq using absolute names (extending the *`dirpath`* prefix) (see Section *Qualified names*). Note that only those subdirectories and files which obey the lexical conventions of what is an *`ident`* are taken into account. Conversely, the underlying file systems or operating systems may be more restrictive than Coq. While Linux's ext4 file system supports any Coq recursive layout (within the limit of 255 bytes per filename), the default on NTFS (Windows) or HFS+ (MacOS X) file systems is on the contrary to disallow two files differing only in the case in the same directory.

See also:

Section *Names of libraries*.

-R *directory dirpath* Do as `-Q` *directory dirpath* but make the subdirectory structure of *directory* recursively visible so that the recursive contents of physical *directory* is available from Coq using short or partially qualified names.

See also:

Section *Names of libraries*.

-top *dirpath* Set the toplevel module name to *`dirpath`* instead of `Top`. Not valid for `coqc` as the toplevel module name is inferred from the name of the output file.

-exclude-dir *directory* Exclude any subdirectory named *directory* while processing options such as -R and -Q. By default, only the conventional version control management directories named CVS and_darcs are excluded.

-nois Start from an empty state instead of loading the Init.Prelude module.

-init-file *file* Load *file* as the resource file instead of loading the default resource file from the standard configuration directories.

-q Do not to load the default resource file.

-load-ml-source *file* Load the OCaml source file *file*.

-load-ml-object *file* Load the OCaml object file *file*.

-l *file*, -load-vernac-source *file* Load and execute the Coq script from *file.v*.

-lv *file*, -load-vernac-source-verbose *file* Load and execute the Coq script from *file.v*. Write its contents to the standard output as it is executed.

-load-vernac-object *qualid* Load Coq compiled library *`qualid`*. This is equivalent to running *`Require`* `qualid`.

-ri *qualid*, -require-import *qualid* Load Coq compiled library *`qualid`* and import it. This is equivalent to running *`Require Import qualid`*.

-re *qualid*, -require-export *qualid* Load Coq compiled library *`qualid`* and transitively import it. This is equivalent to running *`Require Export qualid`*.

-rifrom *dirpath qualid*, -require-import-from *dirpath qualid* Load Coq compiled library *`qualid`* and import it. This is equivalent to running `From` *`dirpath`* *`Require Import qualid`*.

-refrom *dirpath qualid*, -require-export-from *dirpath qualid* Load Coq compiled library *`qualid`* and transitively import it. This is equivalent to running `From` *`dirpath`* *`Require Export qualid`*.

-require *qualid* Deprecated; use `-ri` *qualid*.

-batch Exit just after argument parsing. Available for `coqtop` only.

-verbose Output the content of the input file as it is compiled. This option is available for `coqc` only.

-vos Indicate Coq to skip the processing of opaque proofs (i.e., proofs ending with `Qed` or `Admitted`), output a `.vos` files instead of a `.vo` file, and to load `.vos` files instead of `.vo` files when interpreting `Require` commands.

-vok Indicate Coq to check a file completely, to load `.vos` files instead of `.vo` files when interpreting `Require` commands, and to output an empty `.vok` files upon success instead of writing a `.vo` file.

-w (all|none|w_1,...,w) Configure the display of warnings. This option expects all, none or a comma-separated list of warning names or categories (see Section *Controlling display*).

-color (on|off|auto) *Coqtop only.* Enable or disable color output. Default is auto, meaning color is shown only if the output channel supports ANSI escape sequences.

-diffs (on|off|removed) *Coqtop only.* Controls highlighting of differences between proof steps. `on` highlights added tokens, `removed` highlights both added and removed tokens. Requires that `-color` is enabled. (see Section *Showing differences between proof steps*).

-beautify Pretty-print each command to *file.beautified* when compiling *file.v*, in order to get old-fashioned syntax/definitions/notations.

-emacs, -ide-slave Start a special toplevel to communicate with a specific IDE.

-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.

-type-in-type Collapse the universe hierarchy of Coq.

> **Warning:** This makes the logic inconsistent.

-mangle-names *ident* *Experimental.* Do not depend on this option. Replace Coq's auto-generated name scheme with names of the form *ident0*, *ident1*, etc. Within Coq, the *`Mangle Names`* flag turns this behavior on, and the *`Mangle Names Prefix`* option sets the prefix to use. This feature is intended to be used as a linter for developments that want to be robust to changes in the auto-generated name scheme. The options are provided to facilitate tracking down problems.

-set *string* Enable flags and set options. *string* should be `Option Name=value`, the value is interpreted according to the type of the option. For flags `Option Name` is equivalent to `Option Name=true`. For instance `-set "Universe Polymorphism"` will enable *`Universe Polymorphism`*. Note that the quotes are shell syntax, Coq does not see them.

-unset *string* As `-set` but used to disable options and flags.

-compat *version* Attempt to maintain some backward-compatibility with a previous version.

-dump-glob *file* Dump references for global names in file *file* (to be used by coqdoc, see *Documenting Coq files with coqdoc*). By default, if *file.v* is being compiled, *file.glob* is used.

-no-glob Disable the dumping of references for global names.

-image *file* Set the binary image to be used by `coqc` to be *file* instead of the standard one. Not of general use.

-bindir *directory* Set the directory containing Coq binaries to be used by `coqc`. It is equivalent to doing export COQBIN= *directory* before launching `coqc`.

-where Print the location of Coq's standard library and exit.

-config Print the locations of Coq's binaries, dependencies, and libraries, then exit.

-filteropts Print the list of command line arguments that `coqtop` has recognized as options and exit.

-v Print Coq's version and exit.

-list-tags Print the highlight tags known by Coq as well as their currently associated color and exit.

-h, –help Print a short usage and exit.

7.1.4 Compiled interfaces (produced using `-vos`)

Compiled interfaces help saving time while developing Coq formalizations, by compiling the formal statements exported by a library independently of the proofs that it contains.

> **Warning:** Compiled interfaces should only be used for development purposes. At the end of the day, one still needs to proof check all files by producing standard `.vo` files. (Technically, when using `-vos`, fewer universe constraints are collected.) Moreover, this feature is still experimental, it may be subject to change without prior notice.

Principle.

The compilation using `coqc -vos foo.v` produces a file called `foo.vos`, which is similar to `foo.vo` except that all opaque proofs are skipped in the compilation process.

The compilation using `coqc -vok foo.v` checks that the file `foo.v` correctly compiles, including all its opaque proofs. If the compilation succeeds, then the output is a file called `foo.vok`, with empty contents. This file is only a placeholder indicating that `foo.v` has been successfully compiled. (This placeholder is useful for build systems such as `make`.)

When compiling a file `bar.v` that depends on `foo.v` (for example via a `Require Foo.` command), if the compilation command is `coqc -vos bar.v` or `coqc -vok bar.v`, then the file `foo.vos` gets loaded (instead of `foo.vo`). A special case is if file `foo.vos` exists and has empty contents, and `foo.vo` exists, then `foo.vo` is loaded.

Appart from the aforementioned case where `foo.vo` can be loaded in place of `foo.vos`, in general the `.vos` and `.vok` files live totally independently from the `.vo` files.

Dependencies generated by ‘‘coq_makefile‘‘.

The files `foo.vos` and `foo.vok` both depend on `foo.v`.

Furthermore, if a file `foo.v` requires `bar.v`, then `foo.vos` and `foo.vok` also depend on `bar.vos`.

Note, however, that `foo.vok` does not depend on `bar.vok`. Hence, as detailed further, parallel compilation of proofs is possible.

In addition, `coq_makefile` generates for a file `foo.v` a target `foo.required_vos` which depends on the list of `.vos` files that `foo.vos` depends upon (excluding `foo.vos` itself). As explained next, the purpose of this target is to be able to request the minimal working state for editing interactively the file `foo.v`.

> **Warning:** When writing a custom build system, be aware that `coqdep` only produces dependencies related to `.vos` and `.vok` if the `-vos` command line flag is passed. This is to maintain compatibility with dune (see ocaml/dune#2642 on github[310]).

[310] https://github.com/ocaml/dune/issues/2842

Typical compilation of a set of file using a build system.

Assume a file `foo.v` that depends on two files `f1.v` and `f2.v`. The command `make foo.required_vos` will compile `f1.v` and `f2.v` using the option `-vos` to skip the proofs, producing `f1.vos` and `f2.vos`. At this point, one is ready to work interactively on the file `foo.v`, even though it was never needed to compile the proofs involved in the files `f1.v` and `f2.v`.

Assume a set of files `f1.v ... fn.v` with linear dependencies. The command `make vos` enables compiling the statements (i.e. excluding the proofs) in all the files. Next, `make -j vok` enables compiling all the proofs in parallel. Thus, calling `make -j vok` directly enables taking advantage of a maximal amount of parallelism during the compilation of the set of files.

Note that this comes at the cost of parsing and typechecking all definitions twice, once for the `.vos` file and once for the `.vok` file. However, if files contain nontrivial proofs, or if the files have many linear chains of dependencies, or if one has many cores available, compilation should be faster overall.

Need for "Proof using"

When a theorem is part of a section, typechecking the statement of this theorem might be insufficient for deducing the type of this statement as of at the end of the section. Indeed, the proof of the theorem could make use of section variables or section hypotheses that are not mentioned in the statement of the theorem.

For this reason, proofs inside section should begin with *`Proof using`* instead of *`Proof`*, where after the `using` clause one should provide the list of the names of the section variables that are required for the proof but are not involved in the typechecking of the statement. Note that it is safe to write `Proof using.` instead of `Proof.` also for proofs that are not within a section.

Warning: `You should use the "Proof using [...]." syntax instead of "Proof." to enable skipping this pro`

> If Coq is invoked using the `-vos` option, whenever it finds the command `Proof.` inside a section, it will compile the proof, that is, refuse to skip it, and it will raise a warning. To disable the warning, one may pass the flag `-w -proof-without-using-in-section`.

Interaction with standard compilation

When compiling a file `foo.v` using `coqc` in the standard way (i.e., without `-vos` nor `-vok`), an empty file `foo.vos` and an empty file `foo.vok` are created in addition to the regular output file `foo.vo`. If `coqc` is subsequently invoked on some other file `bar.v` using option `-vos` or `-vok`, and that `bar.v` requires `foo.v`, if Coq finds an empty file `foo.vos`, then it will load `foo.vo` instead of `foo.vos`.

The purpose of this feature is to allow users to benefit from the `-vos` option even if they depend on libraries that were compiled in the traditional manner (i.e., never compiled using the `-vos` option).

7.1.5 Compiled libraries checker (coqchk)

The `coqchk` command takes a list of library paths as argument, described either by their logical name or by their physical filename, hich must end in `.vo`. The corresponding compiled libraries (`.vo` files) are searched in the path, recursively processing the libraries they depend on. The content of all these libraries is then type checked. The effect of `coqchk` is only to return with normal exit code in case of success, and with positive exit code if an error has been found. Error messages are not deemed to help the user understand what is wrong. In the current version, it does not modify the compiled libraries to mark them as successfully checked.

Note that non-logical information is not checked. By logical information, we mean the type and optional body associated to names. It excludes for instance anything related to the concrete syntax of objects (customized syntax rules, association between short and long names), implicit arguments, etc.

This tool can be used for several purposes. One is to check that a compiled library provided by a third-party has not been forged and that loading it cannot introduce inconsistencies[311]. Another point is to get an even higher level of security. Since `coqtop` can be extended with custom tactics, possibly ill-typed code, it cannot be guaranteed that the produced compiled libraries are correct. `coqchk` is a standalone verifier, and thus it cannot be tainted by such malicious code.

Command-line options `-Q`, `-R`, `-where` and `-impredicative-set` are supported by `coqchk` and have the same meaning as for `coqtop`. As there is no notion of relative paths in object files `-Q` and `-R` have exactly the same meaning.

-norec *module* Check *module* but do not check its dependencies.

-admit *module* Do not check *module* and any of its dependencies, unless explicitly required.

-o At exit, print a summary about the context. List the names of all assumptions and variables (constants without body).

-silent Do not write progress information to the standard output.

Environment variable `$COQLIB` can be set to override the location of the standard library.

The algorithm for deciding which modules are checked or admitted is the following: assuming that `coqchk` is called with argument `M`, option `-norec N`, and `-admit A`. Let us write $\overline{S}$ for the set of reflexive transitive dependencies of set S. Then:

- Modules $C = \overline{M} \backslash \overline{A} \cup M \cup N$ are loaded and type checked before being added to the context.
- And $M \cup N \backslash C$ is the set of modules that are loaded and added to the context without type checking. Basic integrity checks (checksums) are nonetheless performed.

As a rule of thumb, -admit can be used to tell Coq that some libraries have already been checked. So `coqchk A B` can be split in `coqchk A` && `coqchk B -admit A` without type checking any definition twice. Of course, the latter is slightly slower since it makes more disk access. It is also less secure since an attacker might have replaced the compiled library `A` after it has been read by the first command, but before it has been read by the second command.

7.2 Utilities

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

7.2.1 Using Coq as a library

In previous versions, `coqmktop` was used to build custom toplevels - for example for better debugging or custom static linking. Nowadays, the preferred method is to use `ocamlfind`.

The most basic custom toplevel is built using:

```
% ocamlfind ocamlopt -thread -rectypes -linkall -linkpkg \
              -package coq.toplevel \
              topbin/coqtop_bin.ml -o my_toplevel.native
```

For example, to statically link L$_{\text{tac}}$, you can just do:

[311] Ill-formed non-logical information might for instance bind Coq.Init.Logic.True to short name False, so apparently False is inhabited, but using fully qualified names, Coq.Init.Logic.False will always refer to the absurd proposition, what we guarantee is that there is no proof of this latter constant.

```
% ocamlfind ocamlopt -thread -rectypes -linkall -linkpkg \
              -package coq.toplevel,coq.plugins.ltac \
              topbin/coqtop_bin.ml -o my_toplevel.native
```

and similarly for other plugins.

7.2.2 Building a Coq project

As of today it is possible to build Coq projects using two tools:

- coq_makefile, which is distributed by Coq and is based on generating a makefile,
- Dune, the standard OCaml build tool, which, since version 1.9, supports building Coq libraries.

Building a Coq project with coq_makefile

The majority of Coq projects are very similar: a collection of `.v` files and eventually some `.ml` ones (a Coq plugin). The main piece of metadata needed in order to build the project are the command line options to `coqc` (e.g. `-R`, `Q`, `-I`, see *command line options*). Collecting the list of files and options is the job of the `_CoqProject` file.

A simple example of a `_CoqProject` file follows:

```
-R theories/ MyCode
-arg -w
-arg all
theories/foo.v
theories/bar.v
-I src/
src/baz.mlg
src/bazaux.ml
src/qux_plugin.mlpack
```

where options `-R`, `-Q` and `-I` are natively recognized, as well as file names. The lines of the form `-arg foo` are used in order to tell to literally pass an argument `foo` to `coqc`: in the example, this allows to pass the two-word option `-w all` (see *command line options*).

Currently, both CoqIDE and Proof-General (version `4.3pre`) understand `_CoqProject` files and invoke Coq with the desired options.

The `coq_makefile` utility can be used to set up a build infrastructure for the Coq project based on makefiles. The recommended way of invoking `coq_makefile` is the following one:

```
coq_makefile -f _CoqProject -o CoqMakefile
```

Such command generates the following files:

CoqMakefile is a generic makefile for `GNU Make` that provides targets to build the project (both `.v` and `.ml*` files), to install it system-wide in the `coq-contrib` directory (i.e. where Coq is installed) as well as to invoke coqdoc to generate HTML documentation.

CoqMakefile.conf contains make variables assignments that reflect the contents of the `_CoqProject` file as well as the path relevant to Coq.

An optional file `CoqMakefile.local` can be provided by the user in order to extend `CoqMakefile`. In particular one can declare custom actions to be performed before or after the build process. Similarly one can customize the install target or even provide new targets. Extension points are documented in paragraph *CoqMakefile.local*.

The extensions of the files listed in `_CoqProject` is used in order to decide how to build them. In particular:

- Coq files must use the `.v` extension
- OCaml files must use the `.ml` or `.mli` extension
- OCaml files that require pre processing for syntax extensions (like `VERNAC EXTEND`) must use the `.mlg` extension
- In order to generate a plugin one has to list all OCaml modules (i.e. `Baz` for `baz.ml`) in a `.mlpack` file (or `.mllib` file).

The use of `.mlpack` files has to be preferred over `.mllib` files, since it results in a “packed” plugin: All auxiliary modules (as `Baz` and `Bazaux`) are hidden inside the plugin's ”namespace” (`Qux_plugin`). This reduces the chances of begin unable to load two distinct plugins because of a clash in their auxiliary module names.

CoqMakefile.local

The optional file `CoqMakefile.local` is included by the generated file `CoqMakefile`. It can contain two kinds of directives.

Variable assignment

The variable must belong to the variables listed in the `Parameters` section of the generated makefile. Here we describe only few of them.

CAMLPKGS can be used to specify third party findlib packages, and is passed to the OCaml compiler on building or linking of modules. Eg: `-package yojson`.

CAMLFLAGS can be used to specify additional flags to the OCaml compiler, like `-bin-annot` or `-w`....

OCAMLWARN it contains a default of `-warn-error +a-3`, useful to modify this setting; beware this is not recommended for projects in Coq's CI.

COQC, COQDEP, COQDOC can be set in order to use alternative binaries (e.g. wrappers)

COQ_SRC_SUBDIRS can be extended by including other paths in which `*.cm*` files are searched. For example `COQ_SRC_SUBDIRS+=user-contrib/Unicoq` lets you build a plugin containing OCaml code that depends on the OCaml code of `Unicoq`

COQFLAGS override the flags passed to `coqc`. By default `-q`.

COQEXTRAFLAGS extend the flags passed to `coqc`

COQCHKFLAGS override the flags passed to `coqchk`. By default `-silent -o`.

COQCHKEXTRAFLAGS extend the flags passed to `coqchk`

COQDOCFLAGS override the flags passed to `coqdoc`. By default `-interpolate -utf8`.

COQDOCEXTRAFLAGS extend the flags passed to `coqdoc`

Rule extension

The following makefile rules can be extended.

Example

```
pre-all::
        echo "This line is print before making the all target"
install-extra::
        cp ThisExtraFile /there/it/goes
```

pre-all:: run before the `all` target. One can use this to configure the project, or initialize sub modules or check dependencies are met.

post-all:: run after the `all` target. One can use this to run a test suite, or compile extracted code.

install-extra:: run after `install`. One can use this to install extra files.

install-doc:: One can use this to install extra doc.

uninstall::

uninstall-doc::

clean::

cleanall::

archclean::

merlin-hook:: One can append lines to the generated `.merlin` file extending this target.

Timing targets and performance testing

The generated `Makefile` supports the generation of two kinds of timing data: per-file build-times, and per-line times for an individual file.

The following targets and Makefile variables allow collection of per- file timing data:

- `TIMED=1` passing this variable will cause `make` to emit a line describing the user-space build-time and peak memory usage for each file built.

  Note: On `Mac OS`, this works best if you've installed `gnu-time`.

  Example

  For example, the output of `make TIMED=1` may look like this:

  ```
  COQDEP Fast.v
  COQDEP Slow.v
  COQC Slow.v
  Slow (user: 0.34 mem: 395448 ko)
  COQC Fast.v
  Fast (user: 0.01 mem: 45184 ko)
  ```

- `pretty-timed` this target stores the output of `make TIMED=1` into `time-of-build.log`, and displays a table of the times, sorted from slowest to fastest, which is also stored in `time-of-build-pretty.log`. If you want to construct the `log` for targets other than the default one, you can pass them via the variable `TGTS`, e.g., `make pretty-timed TGTS="a.vo b.vo"`.

Note: This target will *append* to the timing log; if you want a fresh start, you must remove the `filetime-of-build.log` or `run make cleanall`.

Example

For example, the output of `make pretty-timed` may look like this:

```
COQDEP Fast.v
COQDEP Slow.v
COQC Slow.v
Slow (user: 0.36 mem: 393912 ko)
COQC Fast.v
Fast (user: 0.05 mem: 45992 ko)
Time     | File Name
--------------------
0m00.41s | Total
--------------------
0m00.36s | Slow
0m00.05s | Fast
```

- **`print-pretty-timed-diff`** this target builds a table of timing changes between two compilations; run `make make-pretty-timed-before` to build the log of the "before" times, and run `make make-pretty-timed-after` to build the log of the "after" times. The table is printed on the command line, and stored in `time-of-build-both.log`. This target is most useful for profiling the difference between two commits in a repository.

Note: This target requires `python` to build the table.

Note: The `make-pretty-timed-before` and `make-pretty-timed-after` targets will *append* to the timing log; if you want a fresh start, you must remove the files `time-of-build-before.log` and `time-of-build-after.log` or run `make cleanall` *before* building either the "before" or "after" targets.

Note: The table will be sorted first by absolute time differences rounded towards zero to a whole-number of seconds, then by times in the "after" column, and finally lexicographically by file name. This will put the biggest changes in either direction first, and will prefer sorting by build-time over subsecond changes in build time (which are frequently noise); lexicographic sorting forces an order on files which take effectively no time to compile.

Example

For example, the output table from `make print-pretty-timed-diff` may look like this:

```
After    | File Name | Before   || Change    | % Change
--------------------------------------------------------
0m00.39s | Total     | 0m00.35s || +0m00.03s | +11.42%
--------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
0m00.37s | Slow      | 0m00.01s || +0m00.36s | +3600.00%
0m00.02s | Fast      | 0m00.34s || -0m00.32s | -94.11%
```

The following targets and Makefile variables allow collection of per- line timing data:

- TIMING=1 passing this variable will cause make to use coqc -time to write to a .v.timing file for each .v file compiled, which contains line-by-line timing information.

  Example

  For example, running make all TIMING=1 may result in a file like this:

  ```
  Chars 0 - 26 [Require~Coq.ZArith.BinInt.] 0.157 secs (0.128u,0.028s)
  Chars 27 - 68 [Declare~Reduction~comp~:=~vm_c...] 0. secs (0.u,0.s)
  Chars 69 - 162 [Definition~foo0~:=~Eval~comp~i...] 0.153 secs (0.136u,0.019s)
  Chars 163 - 208 [Definition~foo1~:=~Eval~comp~i...] 0.239 secs (0.236u,0.s)
  ```

- print-pretty-single-time-diff

  ```
  print-pretty-single-time-diff BEFORE=path/to/file.v.before-timing AFTER=path/to/file.
   ↪v.after-timing
  ```

  this target will make a sorted table of the per-line timing differences between the timing logs in the BEFORE and AFTER files, display it, and save it to the file specified by the TIME_OF_PRETTY_BUILD_FILE variable, which defaults to time-of-build-pretty.log. To generate the .v.before-timing or .v.after-timing files, you should pass TIMING=before or TIMING=after rather than TIMING=1.

  Note: The sorting used here is the same as in the print-pretty-timed-diff target.

  Note: This target requires python to build the table.

  Example

  For example, running print-pretty-single-time-diff might give a table like this:

  ```
  After     | Code                                                | Before    ||␣
   ↪Change    | % Change
  ----------------------------------------------------------------------------------
   ↪--------------
  0m00.50s  | Total                                               | 0m04.17s  || -0m03.
   ↪66s | -87.96%
  ----------------------------------------------------------------------------------
   ↪--------------
  0m00.145s | Chars 069 - 162 [Definition~foo0~:=~Eval~comp~i...] | 0m00.192s || -0m00.
   ↪04s | -24.47%
  0m00.126s | Chars 000 - 026 [Require~Coq.ZArith.BinInt.]        | 0m00.143s || -0m00.
   ↪01s | -11.88%
     N/A    | Chars 027 - 068 [Declare~Reduction~comp~:=~nati...] | 0m00.s    || +0m00.
   ↪00s | N/A
  ```

(continues on next page)

(continued from previous page)

```
0m00.s    | Chars 027 - 068 [Declare~Reduction~comp~:=~vm_c...] |    N/A    || +0m00.
↪00s | N/A
0m00.231s | Chars 163 - 208 [Definition~foo1~:=~Eval~comp~i...] | 0m03.836s || -0m03.
↪60s | -93.97%
```

- **all.timing.diff, path/to/file.v.timing.diff** The `path/to/file.v.timing.diff` target will make a `.v.timing.diff` file for the corresponding `.v` file, with a table as would be generated by the `print-pretty-single-time-diff` target; it depends on having already made the corresponding `.v.before-timing` and `.v.after-timing` files, which can be made by passing `TIMING=before` and `TIMING=after`. The `all.timing.diff` target will make such timing difference files for all of the `.v` files that the `Makefile` knows about. It will fail if some `.v.before-timing` or `.v.after-timing` files don't exist.

  Note: This target requires python to build the table.

Reusing/extending the generated Makefile

Including the generated makefile with an include directive is discouraged. The contents of this file, including variable names and status of rules shall change in the future. Users are advised to include `Makefile.conf` or call a target of the generated Makefile as in `make -f Makefile target` from another Makefile.

One way to get access to all targets of the generated `CoqMakefile` is to have a generic target for invoking unknown targets.

Example

```
# KNOWNTARGETS will not be passed along to CoqMakefile
KNOWNTARGETS := CoqMakefile extra-stuff extra-stuff2
# KNOWNFILES will not get implicit targets from the final rule, and so
# depending on them won't invoke the submake
# Warning: These files get declared as PHONY, so any targets depending
# on them always get rebuilt
KNOWNFILES   := Makefile _CoqProject

.DEFAULT_GOAL := invoke-coqmakefile

CoqMakefile: Makefile _CoqProject
        $(COQBIN)coq_makefile -f _CoqProject -o CoqMakefile

invoke-coqmakefile: CoqMakefile
        $(MAKE) --no-print-directory -f CoqMakefile $(filter-out $(KNOWNTARGETS),$(MAKECMDGOALS))

.PHONY: invoke-coqmakefile $(KNOWNFILES)

####################################################################
##                      Your targets here                         ##
####################################################################

# This should be the last rule, to handle any targets not declared above
%: invoke-coqmakefile
        @true
```

Building a subset of the targets with -j

To build, say, two targets foo.vo and bar.vo in parallel one can use `make only TGTS="foo.vo bar.vo" -j`.

Note: `make foo.vo bar.vo -j` has a different meaning for the make utility, in particular it may build a shared prerequisite twice.

Note: For users of coq_makefile with version < 8.7

- Support for ”subdirectory” is deprecated. To perform actions before or after the build (like invoking `make` on a subdirectory) one can hook in pre-all and post-all extension points.
- `-extra-phony` and `-extra` are deprecated. To provide additional target (`.PHONY` or not) please use `CoqMakefile.local`.

Building a Coq project with Dune

Note: Dune's Coq support is still experimental; we strongly recommend using Dune 2.3 or later.

Note: The canonical documentation for the Coq Dune extension is maintained upstream; please refer to the Dune manual[312] for up-to-date information. This documentation is up to date for Dune 2.3.

Building a Coq project with Dune requires setting up a Dune project for your files. This involves adding a `dune-project` and `pkg.opam` file to the root (`pkg.opam` can be empty or generated by Dune itself), and then providing `dune` files in the directories your `.v` files are placed. For the experimental version ”0.1” of the Coq Dune language, Coq library stanzas look like:

```
(coq.theory
 (name <module_prefix>)
 (package <opam_package>)
 (synopsis <text>)
 (modules <ordered_set_lang>)
 (libraries <ocaml_libraries>)
 (flags <coq_flags>))
```

This stanza will build all `.v` files in the given directory, wrapping the library under `<module_prefix>`. If you declare an `<opam_package>`, an `.install` file for the library will be generated; the optional `(modules <ordered_set_lang>)` field allows you to filter the list of modules, and `(libraries <ocaml_libraries>)` allows the Coq theory depend on ML plugins. For the moment, Dune relies on Coq's standard mechanisms (such as `COQPATH`) to locate installed Coq libraries.

By default Dune will skip `.v` files present in subdirectories. In order to enable the usual recursive organization of Coq projects add

[312] https://dune.readthedocs.io/

```
(include_subdirs qualified)
```

to you `dune` file.

Once your project is set up, `dune build` will generate the `pkg.install` files and all the files necessary for the installation of your project.

Example

A typical stanza for a Coq plugin is split into two parts. An OCaml build directive, which is standard Dune:

```
(library
 (name equations_plugin)
 (public_name equations.plugin)
 (flags :standard -warn-error -3-9-27-32-33-50)
 (libraries coq.plugins.cc coq.plugins.extraction))

(coq.pp (modules g_equations))
```

And a Coq-specific part that depends on it via the `libraries` field:

```
(coq.theory
 (name Equations) ; -R flag
 (package equations)
 (synopsis "Equations Plugin")
 (libraries coq.plugins.extraction equations.plugin)
 (modules :standard \ IdDec NoCycle)) ; exclude some modules that don't build

(include_subdirs qualified)
```

7.2.3 Computing Module dependencies

In order to compute module dependencies (to be used by `make` or `dune`), Coq provides the `coqdep` tool.

`coqdep` computes inter-module dependencies for Coq and OCaml 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`), but also at the command `Declare ML Module`.

Dependencies of OCaml modules are computed by looking at `open` commands and the dot notation *module.value*. However, this is done approximately and you are advised to use `ocamldep` instead for the OCaml module dependencies.

See the man page of `coqdep` for more details and options.

Both Dune and `coq_makefile` use `coqdep` to compute the dependencies among the files part of a Coq project.

7.2.4 Documenting Coq files with coqdoc

coqdoc is a documentation tool for the proof assistant Coq, similar to `javadoc` or `ocamldoc`. The task of coqdoc is

1. to produce a nice LaTeX and/or HTML document from Coq source files, readable for a human and not only for the proof assistant;

2. to help the user navigate his own (or third-party) sources.

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 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 `fun x => u` by writing `[fun x => u]`). Inside quotations, the code is pretty-printed in the same way as it is in code parts.

Preformatted vernacular is enclosed by `[[` and `]]`. The former must be followed by a newline and the latter must follow a newline.

Pretty-printing.

coqdoc uses different faces for identifiers and keywords. The pretty- printing of Coq tokens (identifiers or symbols) can be controlled using one of the following commands:

```
(** printing  *token* %...LATEX...% #...html...# *)
```

or

```
(** printing  *token* $...LATEX math...$ #...html...# *)
```

It gives the LaTeX and HTML texts to be produced for the given Coq token. Either the LaTeX or the HTML rule may be omitted, 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:

`->`	→		`<-`	←		`*`	×
`<=`			`>=`			`=>`	
`<>`			`<->`			`|-`	
`\/`			`/\`			`~`	¬

Any of these can be overwritten or suppressed using the printing commands.

Note: The recognition of tokens is done by a (`ocaml`) lex automaton and thus applies the longest-match rule. For instance, `->~` is recognized as a single token, where Coq sees two tokens. It is the responsibility of the user to insert space between tokens *or* to give pretty-printing rules for the possible combinations, e.g.

```
(** printing ->~ %\ensuremath{\rightarrow\lnot}% *)
```

Sections

Sections are introduced by 1 to 4 asterisks at the beginning of a line followed by a space and the title of the section. One asterisk is a section, two a subsection, etc.

Example

```
(** * Well-founded relations

    In this section, we introduce...  *)
```

Lists.

List items are introduced by a leading dash. coqdoc uses whitespace to determine the depth of a new list item and which text belongs in which list items. A list ends when a line of text starts at or before the level of indenting of the list's dash. A list item's dash must always be the first non-space character on its line (so, in particular, a list can not begin on the first line of a comment - start it on the second line instead).

Example

```
We go by induction on [n]:
- If [n] is 0...
- If [n] is [S n'] we require...

  two paragraphs of reasoning, and two subcases:

  - In the first case...
  - In the second case...

So the theorem holds.
```

Rules.

More than 4 leading dashes produce a horizontal rule.

Emphasis.

Text can be italicized by enclosing it in underscores. A non-identifier character must precede the leading underscore and follow the trailing underscore, so that uses of underscores in names aren't mistaken for emphasis. Usually, these are spaces or punctuation.

```
This sentence contains some _emphasized text_.
```

Escaping 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.

Note: to simply output the characters `$`, `%` and `#` and escaping their escaping role, these characters must be doubled.

Verbatim

Verbatim material is introduced by a leading `<<` and closed by `>>` at the beginning of a line.

Example

```
Here is the corresponding caml code:
<<
  let rec fact n =
    if n <= 1 then 1 else n * fact (n-1)
>>
```

Hyperlinks

Hyperlinks can be inserted into the HTML output, so that any identifier is linked to the place of its definition.

`coqc file.v` automatically dumps localization information in `file.glob` or appends it to a file specified using the option `--dump-glob file`. Take care of erasing this global file, if any, when starting the whole compilation process.

Then invoke coqdoc or `coqdoc --glob-from file` to tell coqdoc to look for name resolutions in the file `file` (it will look in `file.glob` by default).

Identifiers from the Coq standard library are linked to the Coq website 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.

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 format. 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 using the option `-o`. The order of files on the command line is kept in the final document. LaTeX files given on the command line are copied 'as is' in the final document . DVI and PostScript can be produced directly with the options `-dvi` and `-ps` respectively.

TEXmacs output To translate the input files to TEXmacs format, to be used by the TEXmacs Coq interface.

Command line options

Overall options

–**HTML** Select a HTML output.

–**LaTeX** Select a LaTeX output.

–**dvi** Select a DVI output.

–**ps** Select a PostScript output.

–**texmacs** Select a TEXmacs output.

–**stdout** Write output to stdout.

-o file, –output file Redirect the output into the file 'file' (meaningless with `-html`).

-d dir, –directory dir Output files into directory 'dir' instead of the current directory (option `-d` does not change the filename specified with the option `-o`, if any).

–**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 filenames to be processed from the file 'file' as if they were given on the command line. Useful for program sources split up into 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

The 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`.

–index string Make the filename of the index string instead of "index". Useful since "index.html" is special.

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`.

–toc-depth int Only include headers up to depth `int` in the table of contents.

Hyperlink 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.

–external url coqdir Use given URL for linking references whose name starts with prefix `coqdir`.

–coqlib url Set base URL for the Coq standard library (default is http://coq.inria.fr/library/). This is equivalent to `--external url Coq`.

-R dir coqdir Recursively map physical directory dir to Coq logical directory `coqdir` (similarly to Coq option `-R`).

-Q dir coqdir Map physical directory dir to Coq logical directory `coqdir` (similarly to Coq option `-Q`).

Note: options `-R` and `-Q` only have effect on the files *following* them on the command line, so you will probably need to put this option first.

Title options

-s , –short Do not insert titles for the files. The default behavior is to insert a title like "Library Foo" for each file.

–lib-name string Print "string Foo" instead of "Library Foo" in titles. For example "Chapter" and "Module" are reasonable choices.

–no-lib-name Print just "Foo" instead of "Library Foo" in titles.

–lib-subtitles Look for library subtitles. When enabled, the beginning of each file is checked for a comment of the form:

```
(** * ModuleName : text *)
```

where `ModuleName` must be the name of the file. If it is present, the text is used as a subtitle for the module in appropriate places.

-t string, –title string Set the document title.

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).

There are a few options that control the parsing of comments:

–parse-comments Parse regular comments delimited by `(*` and `*)` as well. They are typeset inline.

–plain-comments Do not interpret comments, simply copy them as plain-text.

–interpolate Use the globalization information to typeset identifiers appearing in Coq escapings inside comments.

Language options

The default behavior is to assume ASCII 7 bit input files.

-latin1, –latin1 Select ISO-8859-1 input files. It is equivalent to –inputenc latin1 –charset iso-8859-1.

-utf8, –utf8 Set –inputenc utf8x for LaTeX output and–charset utf-8 for HTML output. Also use Unicode replacements for a couple of standard plain ASCII notations such as → for `->` and for `forall`. LaTeX UTF-8 support can be found at http://www.ctan.org/pkg/unicode. For the interpretation of Unicode characters by LaTeX, extra packages which coqdoc does not provide by default might be required, such as textgreek for some Greek letters or `stmaryrd` for some mathematical symbols. If a Unicode character is missing an interpretation in the utf8x input encoding, add `\DeclareUnicodeCharacter{code}{LATEX-interpretation}`. Packages and declarations can be added with option `-p`.

–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.

The coqdoc LaTeX style file

In case you choose to produce a document without the default LaTeX preamble (by using option `--no-preamble`), then you must insert into your own preamble the command

```
\usepackage{coqdoc}
```

The package optionally takes the argument [color] to typeset identifiers with colors (this requires the xcolor package).

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.

7.2.5 Embedded Coq phrases inside LaTeX documents

When writing documentation about a proof development, one may want to insert Coq phrases inside a LaTeX document, possibly together with the corresponding answers of the system. We provide a mechanical way to process such Coq phrases embedded in LaTeX files: the coq-tex filter. This filter extracts Coq phrases embedded in LaTeX files, evaluates them, and insert the outcome of the evaluation after each phrase.

Starting with a file file.tex containing Coq phrases, the coq-tex filter produces a file named file.v.tex with the Coq outcome.

There are options to produce the Coq parts in smaller font, italic, between horizontal rules, etc. See the man page of coq-tex for more details.

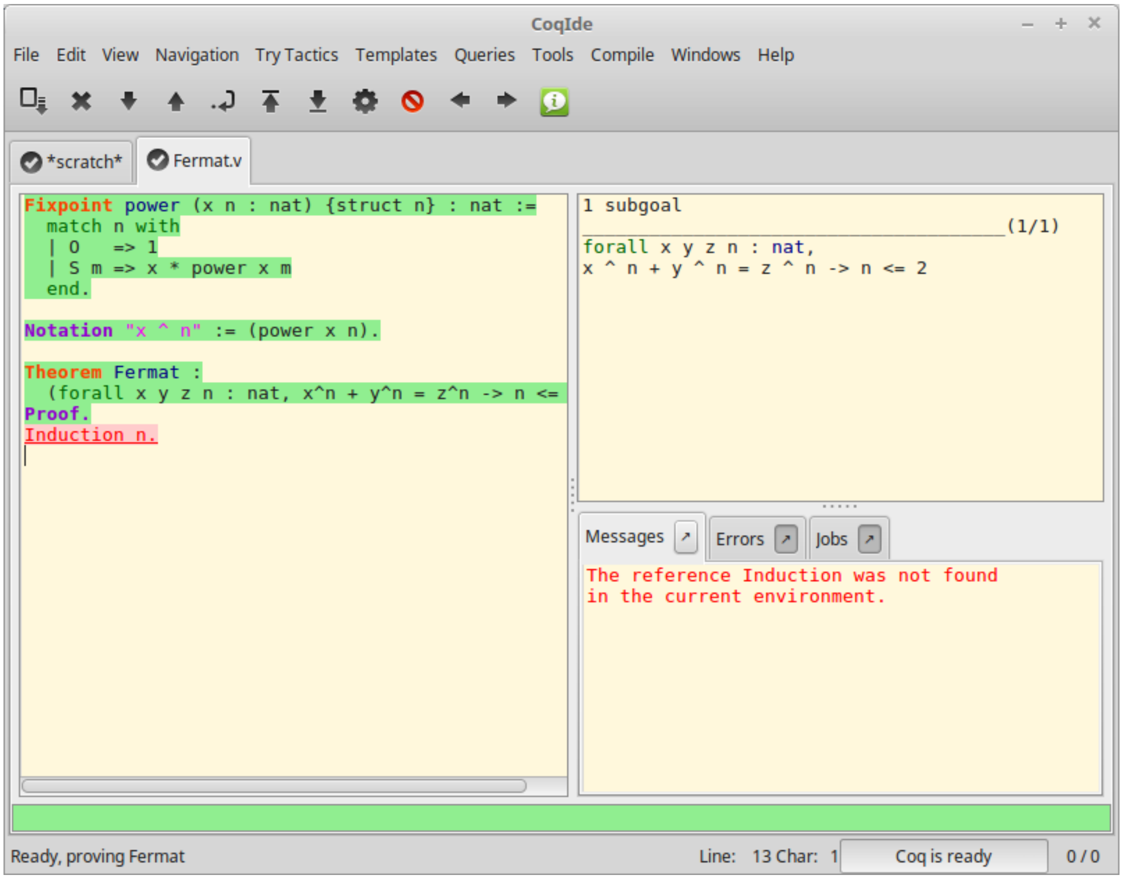
7.2.6 Man pages

There are man pages for the commands coqdep and coq-tex. Man pages are installed at installation time (see installation instructions in file INSTALL, step 6).

7.3 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 filename as argument, another buffer displaying the contents of that file. Additionally, coqide accepts the same options as coqtop, given in *The Coq commands*, the ones having obviously no meaning for CoqIDE being ignored.



```
Fixpoint power (x n : nat) {struct n} : nat :=
  match n with
  | 0   => 1
  | S m => x * power x m
  end.

Notation "x ^ n" := (power x n).

Theorem Fermat :
  (forall x y z n : nat, x^n + y^n = z^n -> n <=
Proof.
Induction n.
```

```
1 subgoal
______________________________________(1/1)
forall x y z n : nat,
x ^ n + y ^ n = z ^ n -> n <= 2
```

```
The reference Induction was not found
in the current environment.
```

A sample CoqIDE main screen, while navigating into a file `Fermat.v`, is shown in the figure *CoqIDE main screen*. 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 be proven 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.

7.3.1 Managing files and buffers, basic editing

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 background in the *processed* color (green by default), 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.

7.3.2 Interactive navigation into Coq scripts

The running buffer is the one where navigation takes place. The toolbar offers 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 background with the processed color. If that command fails, the error message is displayed in the message window, and the location of the error is emphasized by an underline in the error foreground color (red by default).

In the figure *CoqIDE main screen*, the running buffer is `Fermat.v`, all commands until the `Theorem` have been already executed, and the user tried to go forward executing `Induction n`. That command failed because no such tactic exists (names of standard tactics are written in lowercase), and the failing command is underlined.

Notice that the processed 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.

There are two additional buttons for navigation within the running buffer. The ”down” button with a line goes directly to the end; the ”up” button with a line goes back to the beginning. The handling of errors when using the go-to-the-end button depends on whether Coq is running in asynchronous mode or not (see Chapter *Asynchronous and Parallel Proof Processing*). If it is not running in that mode, execution stops as soon as an error is found. Otherwise, execution continues, and the error is marked with an underline in the error foreground color, with a background in the error background color (pink by default). The same characterization of error-handling applies when running several commands using the ”goto” button.

If you ever try to execute a command that runs for a long time and would like to abort it before it terminates, you may use the interrupt button (the white cross on a red circle).

There are other buttons on the CoqIDE toolbar: a button to save the running buffer; a button to close the current buffer (an ”X”); buttons to switch among buffers (left and right arrows); an ”information” button; and a ”gears” button.

The ”gears” button submits proof terms to the Coq kernel for type checking. When Coq uses asynchronous processing (see Chapter *Asynchronous and Parallel Proof Processing*), proofs may have been completed without kernel-checking of generated proof terms. The presence of unchecked proof terms is indicated by `Qed` statements that have a subdued *being-processed* color (light blue by default), rather than the processed color, though their preceding proofs have the processed color.

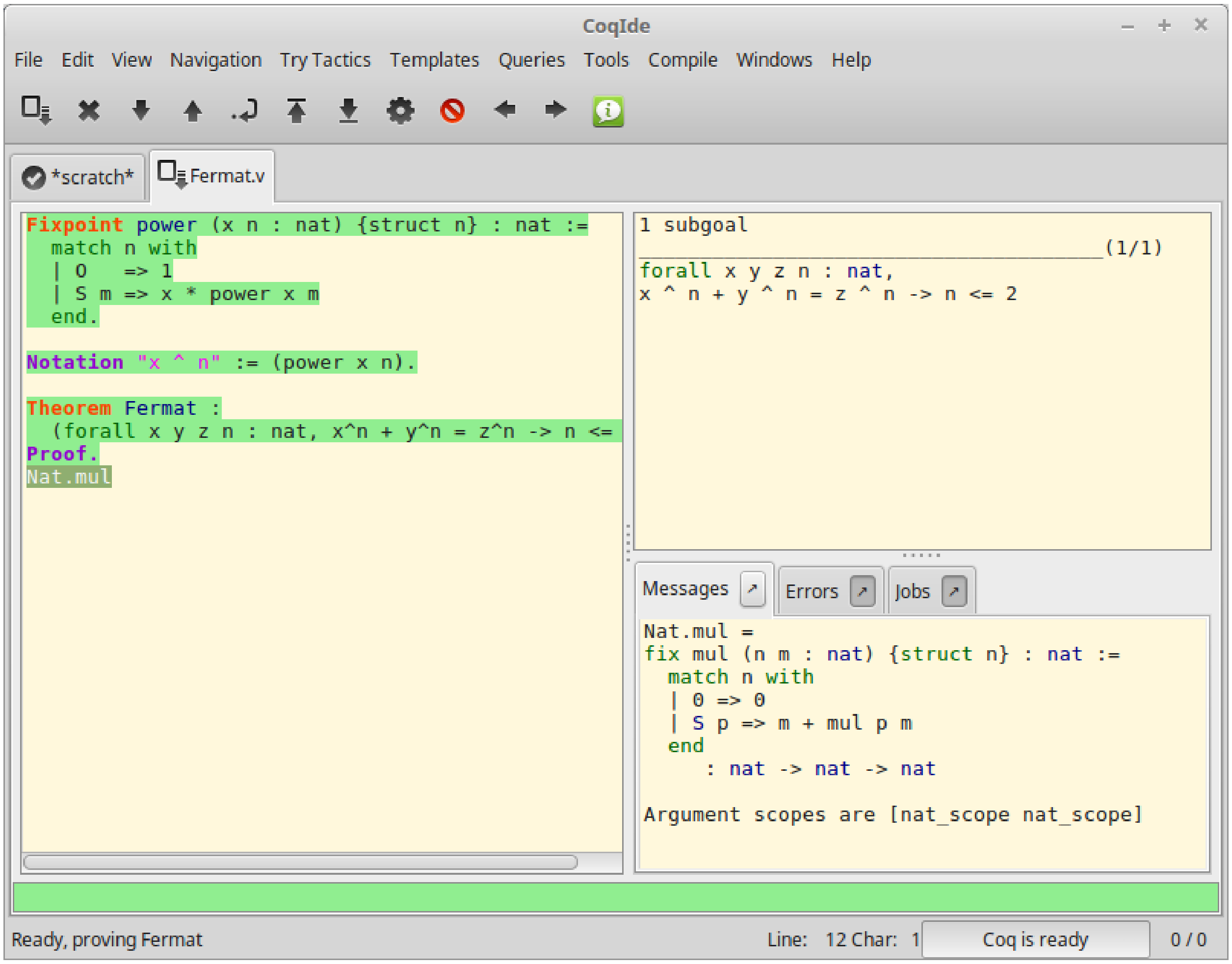
Notice that for all these buttons, except for the ”gears” button, their operations are also available in the menu, where their keyboard shortcuts are given.

7.3.3 Vernacular commands, templates

The Templates menu allows using shortcuts to insert vernacular commands. This is a nice way to proceed if you are not sure of the syntax of the command you want.

Moreover, from this menu you can automatically insert templates of complex commands like `Fixpoint` that you can conveniently fill afterwards.

7.3.4 Queries



```
Fixpoint power (x n : nat) {struct n} : nat :=
  match n with
  | 0   => 1
  | S m => x * power x m
  end.

Notation "x ^ n" := (power x n).

Theorem Fermat :
  (forall x y z n : nat, x^n + y^n = z^n -> n <=
Proof.
Nat.mul
```

```
1 subgoal
______________________________________(1/1)
forall x y z n : nat,
x ^ n + y ^ n = z ^ n -> n <= 2
```

```
Nat.mul =
fix mul (n m : nat) {struct n} : nat :=
  match n with
  | 0 => 0
  | S p => m + mul p m
  end
     : nat -> nat -> nat

Argument scopes are [nat_scope nat_scope]
```

We call *query* any vernacular command that does not change the current state, such as `Check`, `Search`, etc. To run such commands interactively, without writing them in scripts, CoqIDE offers a *query pane*. The query pane can be displayed on demand by using the `View` menu, or using the shortcut `F1`. Queries can also be performed by selecting a particular phrase, then choosing an item from the `Queries` menu. The response then appears in the message window. The image above shows the result after selecting of the phrase `Nat.mul` in the script window, and choosing `Print` from the `Queries` menu.

7.3.5 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`.

7.3.6 Customizations

You may customize your environment using the 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 and third sections are for controlling colors and style of the three main buffers. A predefined Coq highlighting style as well as standard GtkSourceView styles are available. Other styles can be added e.g. in `$HOME/.local/share/gtksourceview-3.0/styles/` (see the general documentation about GtkSourceView for the various possibilities). Note that the style of the rest of graphical part of Coqide is not under the control of GtkSourceView but of GTK+ and governed by files such as `settings.ini` and `gtk.css` in `$XDG_CONFIG_HOME/gtk-3.0` or files in `$HOME/.themes/NameOfTheme/gtk-3.0`, as well as the environment variable `GTK_THEME` (search on internet for the various possibilities).

The fourth section is for customizing the editor. It includes in particular the ability to activate an Emacs mode named micro-Proof-General (use the Help menu to know more about the available bindings).

The next 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 prompted to choose to either discard your changes or not. The File charset encoding choice is described below in *Character encoding for saved files*.

The `Externals` section allows customizing the external commands for compilation, printing, web browsing. In the browser command, you may use `%s` to denote the URL to open, for example: `firefox -remote "OpenURL(%s)"`.

Notice that these settings are saved in the file `coqiderc` in the `coq` subdirectory of the user configuration directory which is the value of `$XDG_CONFIG_HOME` if this environment variable is set and which otherwise is `$HOME/.config/`.

A GTK+ accelerator keymap is saved under the name `coqide.keys` in the same `coq` subdirectory of the user configuration directory. It is not recommended to edit this file 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. If your system does not allow it, you may still edit this configuration file by hand, but this is more involved.

7.3.7 Using Unicode symbols

CoqIDE is based on GTK+ and inherits from it support for Unicode in its text windows. Consequently a large set of symbols is available for notations. Furthermore, CoqIDE conveniently provides a simple way to input Unicode characters.

Displaying Unicode symbols

You just need to define suitable notations as described in the chapter *Syntax extensions and interpretation scopes*. For example, to use the mathematical symbols   and , you may define:

```
Notation "  x .. y , P" := (forall x, .. (forall y, P) ..)
  (at level 200, x binder, y binder, right associativity)
  : type_scope.
Notation "  x .. y , P" := (exists x, .. (exists y, P) ..)
  (at level 200, x binder, y binder, right associativity)
  : type_scope.
```

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 Import Unicode.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 choosing whether GTK+ should use antialiased fonts or not, by setting the environment variable `GDK_USE_XFT` to 1 or 0 respectively.

Bindings for input of Unicode symbols

CoqIDE supports a builtin mechanism to input non-ASCII symbols. For example, to input π, it suffices to type `\pi` then press the combination of key `Shift+Space` (default key binding). Often, it suffices to type a prefix of the latex token, e.g. typing `\p` then `Shift+Space` suffices to insert a π.

For several symbols, ASCII art is also recognized, e.g. `\->` for a right arrow, or `\>=` for a greater than or equal sign.

A larger number of latex tokens are supported by default. The full list is available here: https://github.com/coq/coq/blob/master/ide/default_bindings_src.ml

Custom bindings may be added, as explained further on.

Note: It remains possible to input non-ASCII symbols using system-wide approaches independent of CoqIDE.

Adding custom bindings

To extend the default set of bindings, create a file named `coqide.bindings` and place it in the same folder as `coqide.keys`. This would be the folder `$XDG_CONFIG_HOME/coq`, defaulting to `~/.config/coq` if `XDG_CONFIG_HOME` is unset. The file `coqide.bindings` should contain one binding per line, in the form `\key value`, followed by an optional priority integer. (The key and value should not contain any space character.)

Example

Here is an example configuration file:

```
\par ||
\pi π 1
\le   1
\lambda λ 2
\lambdas λs
```

Above, the priority number 1 on `\pi` indicates that the prefix `\p` should resolve to `\pi`, and not to something else (e.g. `\par`). Similarly, the above settings ensure than `\l` resolves to `\le`, and that `\la` resolves to `\lambda`.

It can be useful to work with per-project binding files. For this purpose CoqIDE accepts a command line argument of the form `-unicode-bindings file1,file2,...,fileN`. Each of the file tokens provided may consists of one of:

- a path to a custom bindings file,

- the token `default`, which resolves to the default bindings file,
- the token `local`, which resolves to the `coqide.bindings` file stored in the user configuration directory.

Warning: If a filename other than the first one includes a ”~” to refer to the home directory, it won’t be expanded properly. To work around that issue, one should not use comas but instead repeat the flag, in the form: `-unicode-bindings file1 .. -unicode-bindings fileN`.

Note: If two bindings for a same token both have the same priority value (or both have no priority value set), then the binding considered is the one from the file that comes first on the command line.

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.

CHAPTER EIGHT

ADDENDUM

8.1 Extended pattern matching

Authors Cristina Cornes and Hugo Herbelin

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

8.1.1 Patterns

The full syntax of `match` is presented in section *Terms*. Identifiers in patterns are either constructor names or variables. Any identifier that is not the constructor of an inductive or co-inductive 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 x` where `c` is a constructor symbol and x is a linear vector of (distinct) variables, it is called *simple*: it is the kind of pattern recognized by the basic version of match. On the opposite, if it is a variable `x` or has the form `c p` with `p` not only made of variables, the pattern is called *nested*.

A variable pattern matches any value, and the identifier is bound to that value. The pattern "`_`" (called "don't care" or "wildcard" symbol) also matches any value, but does not bind anything. It may occur an arbitrary number of times in a pattern. Alias patterns written (*`pattern`* `as` *`ident`*) are also accepted. This pattern matches the same values as *`pattern`* does and *`ident`* is bound to the matched value. A pattern of the form *`pattern`* `|` *`pattern`* is called disjunctive. A list of patterns separated with commas is also considered as a pattern and is called *multiple pattern*. However multiple patterns can only occur at the root of pattern matching equations. Disjunctions of *multiple patterns* are allowed though.

Since extended `match` expressions are compiled into the primitive ones, the expressiveness of the theory remains the same. Once parsing has finished only simple patterns remain. The original nesting of the `match` expressions is recovered at printing time. An easy way to see the result of the expansion is to toggle off the nesting performed at printing (use here *`Printing Matching`*), then by printing the term with *`Print`* if the term is a constant, or using the command *`Check`*.

The extended `match` still accepts an optional *elimination predicate* given after the keyword `return`. Given a pattern matching expression, if all the right-hand-sides of `=>` 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 basicmatch.

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:

```
Fixpoint max (n m:nat) {struct m} : nat :=
  match n with
  | O => m
```

(continues on next page)

(continued from previous page)

```
  | S n' => match m with
            | O => S n'
            | S m' => S (max n' m')
            end
  end.
```

8.1.2 Multiple patterns

Using multiple patterns in the definition of `max` lets us write:

```
Fixpoint max (n m:nat) {struct m} : nat :=
    match n, m with
    | O, _ => m
    | S n', O => S n'
    | S n', S m' => S (max n' m')
    end.
```

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.

```
Check (fun x:nat => match x return nat with
                    | y => y
                    end).
    fun x : nat => x
         : nat -> nat
```

8.1.3 Aliasing subpatterns

We can also use `as` *ident* to associate a name to a sub-pattern:

```
Fixpoint max (n m:nat) {struct n} : nat :=
  match n, m with
  | O, _ => m
  | S n' as p, O => p
  | S n', S m' => S (max n' m')
  end.
```

8.1.4 Nested patterns

Here is now an example of nested patterns:

```
Fixpoint even (n:nat) : bool :=
  match n with
  | O => true
  | S O => false
  | S (S n') => even n'
  end.
```

This is compiled into:

```
Unset Printing Matching.
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
      end
         : nat -> bool

    Arguments even _%nat_scope
```

In the previous examples patterns do not conflict with, but sometimes it is comfortable to write patterns that admit a non trivial superposition. Consider the boolean function `lef` that given two natural numbers yields `true` if the first one is less or equal than the second one and `false` otherwise. We can write it as follows:

```
Fixpoint lef (n m:nat) {struct m} : bool :=
  match n, m with
  | O, x => true
  | x, O => false
  | S n, S m => lef n m
  end.
```

Note that the first and the second multiple pattern overlap because the couple of values `O O` matches both. Thus, what is the result of the function on those values? To eliminate ambiguity we use the *textual priority rule:* we consider patterns to be ordered from top to bottom. A value is matched by the pattern at the ith row if and only if it is not matched by some pattern from a previous row. Thus in the example, `O O` is matched by the first pattern, and so `(lef O O)` yields true.

Another way to write this function is:

```
Fixpoint lef (n m:nat) {struct m} : bool :=
  match n, m with
  | O, x => true
  | S n, S m => lef n m
  | _, _ => false
  end.
```

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:

```
Fail Check (fun x:nat =>
              match x with
              | O => true
              | S _ => false
              | x => true
              end).
    The command has indeed failed with message:
    Pattern "x" is redundant in this clause.
```

8.1.5 Disjunctive patterns

Multiple patterns that share the same right-hand-side can be factorized using the notation *mult_pattern*+|. For instance, max can be rewritten as follows:

```
Fixpoint max (n m:nat) {struct m} : nat :=
  match n, m with
  | S n', S m' => S (max n' m')
  | 0, p | p, 0 => p
  end.
```

Similarly, factorization of (not necessarily multiple) patterns that share the same variables is possible by using the notation *pattern*+|. Here is an example:

```
Definition filter_2_4 (n:nat) : nat :=
  match n with
  | 2 as m | 4 as m => m
  | _ => 0
  end.
```

Nested disjunctive patterns are allowed, inside parentheses, with the notation (*pattern*+|), as in:

```
Definition filter_some_square_corners (p:nat*nat) : nat*nat :=
  match p with
  | ((2 as m | 4 as m), (3 as n | 5 as n)) => (m,n)
  | _ => (0,0)
  end.
```

8.1.6 About patterns of parametric types

Parameters in patterns

When matching objects of a parametric type, parameters do not bind in patterns. They must be substituted by "_". Consider for example the type of polymorphic lists:

```
Inductive List (A:Set) : Set :=
| nil : List A
| cons : A -> List A -> List A.
```

We can check the function *tail*:

```
Check
  (fun l:List nat =>
     match l with
     | nil _ => nil nat
     | cons _ _ l' => l'
     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:

```
Fail Check
  (fun l:List nat =>
     match l with
     | nil A => nil nat
     | cons A _ l' => l'
     end).
    The command has indeed failed with message:
    The parameters do not bind in patterns; they must be replaced by '_'.
```

Flag: Asymmetric Patterns

This flag (off by default) removes parameters from constructors in patterns:

```
Set Asymmetric Patterns.
Check (fun l:List nat =>
  match l with
  | nil => nil _
  | cons _ l' => l'
  end).
    fun l : List nat => match l with
                        | @nil _ => nil nat
                        | @cons _ _ l' => l'
                        end
         : List nat -> List nat

Unset Asymmetric Patterns.
```

8.1.7 Implicit arguments in patterns

By default, implicit arguments are omitted in patterns. So we write:

```
Arguments nil {A}.
Arguments cons [A] _ _.
Check
  (fun l:List nat =>
     match l with
     | nil => nil
     | cons _ l' => l'
     end).
    fun l : List nat => match l with
                        | nil => nil
                        | cons _ l' => l'
                        end
         : List nat -> List nat
```

But the possibility to use all the arguments is given by "@" implicit explicitations (as for terms, see *Explicit applications*).

```
Check
  (fun l:List nat =>
     match l with
     | @nil _ => @nil nat
     | @cons _ _ l' => l'
     end).
    fun l : List nat => match l with
                        | nil => nil
                        | cons _ l' => l'
```

(continues on next page)

(continued from previous page)

```
                        end
         : List nat -> List nat
```

8.1.8 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 types. Consider the type `listn` of lists of a certain length:

```
Inductive listn : nat -> Set :=
| niln : listn 0
| consn : forall n:nat, nat -> listn n -> listn (S n).
```

8.1.9 Understanding dependencies in patterns

We can define the function length over `listn` by:

```
Definition length (n:nat) (l:listn n) := n.
```

Just for illustrating pattern matching, we can define it by case analysis:

```
Definition length (n:nat) (l:listn n) :=
  match l with
  | niln => 0
  | consn n _ _ => S n
  end.
```

We can understand the meaning of this definition using the same notions of usual pattern matching.

8.1.10 When the elimination predicate must be provided

Dependent pattern matching

The examples given so far do not need an explicit elimination predicate because all the right hand sides have the same type and Coq succeeds to synthesize it. Unfortunately when dealing with dependent patterns it often happens that we need to write cases where the types of the right hand sides are different instances of the elimination predicate. The function `concat` for `listn` is an example where the branches have different types and we need to provide the elimination predicate:

```
Fixpoint concat (n:nat) (l:listn n) (m:nat) (l':listn m) {struct l} :
 listn (n + m) :=
  match l in listn n return listn (n + m) with
  | niln => l'
  | consn n' a y => consn (n' + m) a (concat n' y m l')
  end.
```

The elimination predicate is `fun (n:nat) (l:listn n) => listn (n+m)`. In general if `m` has type `(I q1 … qr t1 … ts)` where `q1, …, qr` are parameters, the elimination predicate should be of the form `fun y1 … ys x : (I q1 … qr y1 … ys ) => Q`.

In the concrete syntax, it should be written : `match m as x in (I _ … _ y1 … ys) return Q with … 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 `_`.

Multiple dependent pattern matching

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 types 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` clause and an `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:

```
Fixpoint concat (n:nat) (l:listn n) (m:nat) (l':listn m) {struct l} :
 listn (n + m) :=
  match l in listn n, l' return listn (n + m) with
  | niln, x => x
  | consn n' a y, x => consn (n' + m) a (concat n' y m x)
  end.
```

Even without real matching over the second term, this construction can be used to keep types linked. If `a` and `b` are two `listn` of the same length, by writing

```
Check (fun n (a b: listn n) =>
 match a, b with
 | niln, b0 => tt
 | consn n' a y, bS => tt
 end).
```

we have a copy of `b` in type `listn 0` resp. `listn (S n')`.

Patterns in `in`

If the type of the matched term is more precise than an inductive applied to variables, arguments of the inductive in the `in` branch can be more complicated patterns than a variable.

Moreover, constructors whose types do not follow the same pattern will become impossible branches. In an impossible branch, you can answer anything but False_rect unit has the advantage to be subterm of anything.

To be concrete: the `tail` function can be written:

```
Definition tail n (v: listn (S n)) :=
  match v in listn (S m) return listn m with
  | niln => False_rect unit
  | consn n' a y => y
  end.
```

and `tail n v` will be subterm of `v`.

8.1.11 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 a dependent type. An example of a proof written using `match` is given in the description of the tactic *refine*.

For example, we can write the function `buildlist` that given a natural number `n` builds a list of length `n` containing zeros as follows:

```
Fixpoint buildlist (n:nat) : listn n :=
  match n return listn n with
  | O => niln
  | S n => consn n 0 (buildlist n)
  end.
```

We can also use multiple patterns. Consider the following definition of the predicate less-equal `Le`:

```
Inductive LE : nat -> nat -> Prop :=
  | LEO : forall n:nat, LE 0 n
  | LES : forall n m:nat, LE n m -> LE (S n) (S m).
```

We can use multiple patterns to write the proof of the lemma `forall (n m:nat), (LE n m) \/ (LE m n)`:

```
Fixpoint dec (n m:nat) {struct n} : LE n m \/ LE m n :=
  match n, m return LE n m \/ LE m n with
  | O, x => or_introl (LE x 0) (LEO x)
  | x, O => or_intror (LE x 0) (LEO x)
  | S n as n', S m as m' =>
      match dec n m with
      | or_introl h => or_introl (LE m' n') (LES n m h)
      | or_intror h => or_intror (LE n' m') (LES m n h)
      end
  end.
```

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* to build incomplete proofs beginning with a `match` construction.

8.1.12 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

```
Inductive list : nat -> Set :=
| nil : list 0
| 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.

```
Fixpoint length n (l:list n) {struct l} : nat :=
  match l with
  | nil => 0
  | cons n l0 => S (length (2 * n) l0)
  end.
```

But in this example, it is.

```
Fixpoint length' n (l:list n) {struct l} : nat :=
  match l with
  | nil => 0
  | @cons _ m l0 => S (length' m l0)
  end.
```

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

Note: You can only catch let bindings in mode where you bind all variables and so you have to use `@` syntax.

Note: this feature is incoherent with the fact that parameters cannot be caught and consequently is somehow hidden. For example, there is no mention of it in error messages.

8.1.13 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

```
Inductive I : Set :=
  | C1 : nat -> I
  | C2 : I -> I.
Coercion C1 : nat >-> I.

Check (fun x => match x with
                | C2 O => 0
                | _ => 0
                end).
    fun x : I => match x with
                 | C1 _ | _ => 0
                 end
         : I -> nat
```

8.1.14 When does the expansion strategy fail?

The strategy works very like in ML languages when treating patterns of non-dependent types. But there are new cases of failure that are due to the presence of dependencies.

The error messages of the current implementation may be sometimes confusing. When the tactic fails because patterns are somehow incorrect then error messages refer to the initial expression. But the strategy may succeed to build an expression whose sub-expressions are well typed when the whole expression is not. In this situation the message makes reference to the expanded expression. We encourage users, when they have patterns with the same outer constructor in different equations, to name the variable patterns in the same positions with the same name. E.g. to write `(cons n O x) => e1` and `(cons n _ x) => e2` instead of `(cons n O x) => e1` and `(cons n' _ x') => e2`. This helps to maintain certain name correspondence between the generated expression and the original.

Here is a summary of the error messages corresponding to each situation:

Error: **`The constructor`** ***`ident`*** **`expects`** ***`num`*** **`arguments.`**

The variable ident is bound several times in pattern termFound a constructor of inductive type term while a constructor of term is expectedPatterns are incorrect (because constructors are not applied to the correct number of the arguments, because they are not linear or they are wrongly typed).

Error: **`Non exhaustive pattern matching.`**

The pattern matching is not exhaustive.

Error: **`The elimination predicate term should be of arity`** ***`num`*** **`(for non dependent case) or`** ***`num`*** **`(for depend`**

The elimination predicate provided to match has not the expected arity.

Error: **`Unable to infer a match predicate`**

Error: **`Either there is a type incompatibility or the problem involves dependencies.`**

There is a type mismatch between the different branches. The user should provide an elimination predicate.

8.2 Implicit Coercions

Author Amokrane Saïbi

8.2.1 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.

8.2.2 Classes

A class with n parameters is any defined name with a type `forall (`*`ident`*$_1$ `:` *`type`*$_1$`)..(`*`ident`*$_n$`:`*`type`*$_n$`),` *`sort`*. Thus a class with parameters is considered as a single class and not as a family of classes. An object of a class is any term of type *`class term`*$_1$ `..` *`term`*$_n$. In addition to these user-defined classes, we have two built-in classes:

- `Sortclass`, the class of sorts; its objects are the terms whose type is a sort (e.g. `Prop` or `Type`).
- `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 classes is defined as:

```
class  ::=  qualid
            Sortclass
```

```
Funclass
```

8.2.3 Coercions

A name `f` can be declared as a coercion between a source user-defined class `C` with n parameters and a target class `D` if one of these conditions holds:

- `D` is a user-defined class, then the type of `f` must have the form forall (x$_1$:A$_1$)..(x :A)(y:C x$_1$..x), D u$_1$..u where m is the number of parameters of `D`.
- `D` is `Funclass`, then the type of `f` must have the form forall (x$_1$:A$_1$)..(x :A)(y:C x$_1$..x)(x:A), B.
- `D` is `Sortclass`, then the type of `f` must have the form forall (x$_1$:A$_1$)..(x :A)(y:C x$_1$..x), s with `s` a sort.

We then write `f : C >-> D`. The restriction on the type of coercions is called *the uniform inheritance condition.*

Note: The built-in class `Sortclass` can be used as a source class, but the built-in class `Funclass` cannot.

To coerce an object t:C t$_1$..t of `C` towards `D`, we have to apply the coercion `f` to it; the obtained term f t$_1$..t t is then an object of `D`.

8.2.4 Identity Coercions

Identity coercions are special cases of coercions used to go around the uniform inheritance condition. Let `C` and `D` be two classes with respectively `n` and `m` parameters and f:forall (x$_1$:T$_1$)..(x :T)(y:C u$_1$..u), D v$_1$..v a function which does not verify the uniform inheritance condition. To declare `f` as coercion, one has first to declare a subclass `C'` of `C`:

C' := fun (x$_1$:T$_1$)..(x :T) => C u$_1$..u

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

Id_C'_C := fun (x$_1$:T$_1$)..(x :T)(y:C' x$_1$..x) => (y:C u$_1$..u)

We can now declare `f` as coercion from `C'` to `D`, since we can ”cast” its type as forall (x$_1$:T$_1$)..(x :T)(y:C' x$_1$..x),D v$_1$..v .

The identity coercions have a special status: to coerce an object t:C' t$_1$..t of `C'` towards `C`, we do not have to insert explicitly `Id_C'_C` since Id_C'_C t$_1$..t 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 '..u ' where each u ' is the result of the substitution in u of the variables x by t .

8.2.5 Inheritance Graph

Coercions form an inheritance graph with classes as nodes. We call *coercion path* an ordered list of coercions between two nodes of the graph. A class `C` is said to be a subclass of `D` if there is a coercion path in the graph from `C` to `D`; we also say that `C` inherits from `D`. Our mechanism supports multiple inheritance since a class may inherit from several classes, contrary to simple inheritance where a class inherits from at most one class. However there must be at most one path between two classes. If this is not the case, only the *oldest* one is valid and the others are ignored. So the order of declaration of coercions is important.

We extend notations for coercions to coercion paths. For instance `[f`$_1$`;..;f ] : C >-> D` is the coercion path composed by the coercions `f`$_1$`..f .` The application of a coercion path to a term consists of the successive application of its coercions.

8.2.6 Declaring Coercions

Command: `Coercion` *qualid* `:` *class* `>->` *class*

Declares the construction denoted by *qualid* as a coercion between the two given classes.

Error: *qualid* `not declared.`

Error: *qualid* `is already a coercion.`

Error: `Funclass cannot be a source class.`

Error: *qualid* `is not a function.`

Error: `Cannot find the source class of` *qualid*`.`

Error: `Cannot recognize` *class* `as a source class of` *qualid*`.`

Warning: *qualid* `does not respect the uniform inheritance condition.`

Error: `Found target class ... instead of ...`

Warning: `New coercion path ... is ambiguous with existing ...`

When the coercion *qualid* is added to the inheritance graph, new coercion paths which have the same classes as existing ones are ignored. The *Coercion* command tries to check the convertibility of new ones and existing ones. The paths for which this check fails are displayed by a warning in the form `[f`$_1$`;..;f ] : C >-> D`.

The convertibility checking procedure for coercion paths is complete for paths consisting of coercions satisfying the uniform inheritance condition, but some coercion paths could be reported as ambiguous even if they are convertible with existing ones when they have coercions that don't satisfy the uniform inheritance condition.

Variant: `Local Coercion` *qualid* `:` *class* `>->` *class*

Declares the construction denoted by *qualid* as a coercion local to the current section.

Variant: `Coercion` *ident* `:=` *term* *type*?

This defines *ident* just like `Definition` *ident* `:=` `term` *type*?, and then declares *ident* as a coercion between it source and its target.

Variant: `Local Coercion` *ident* `:=` *term* *type*?

This defines *ident* just like `Let` *ident* `:=` *term* *type*?, and then declares *ident* as a coercion between it source and its target.

Assumptions can be declared as coercions at declaration time. This extends the grammar of assumptions from Figure *The Vernacular* as follows:

```
assumption      ::=   assumption_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.

Similarly, 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 *The Vernacular* as follows:

```
inductive     ::=   Inductive ind_body with ... with ind_body
                    CoInductive ind_body with ... with ind_body
ind_body      ::=   ident [ binders ] : term := [[|] constructor | ... | constructor ]
constructor   ::=   ident [ binders ] [:[>] term ]
```

Especially, if the extra `>` is present in a constructor declaration, this constructor is declared as a coercion.

Command: `Identity Coercion` *ident* `:` *class* `>->` *class*

If `C` is the source `class` and `D` the destination, we check that `C` is a constant with a body of the form `fun` $(x_1{:}T_1)..(x\ {:}T\ )$ `=>` `D` $t_1..t$ where `m` is the number of parameters of `D`. Then we define an identity function with type `forall` $(x_1{:}T_1)..(x\ {:}T\ )(y{:}C\ x_1..x\ ),D\ t_1..t$, and we declare it as an identity coercion between `C` and `D`.

Error: *class* `must be a transparent constant.`

Variant: `Local Identity Coercion` *ident* `:` *ident* `>->` *ident*

Same as *Identity Coercion* but locally to the current section.

Variant: `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*. `Identity Coercion Id_`*ident*`_`*ident*' `:` *ident* `>->` *ident*'.

Variant: `Local SubClass` *ident* `:=` *type*

Same as before but locally to the current section.

8.2.7 Displaying Available Coercions

Command: `Print Classes`

Print the list of declared classes in the current context.

Command: `Print Coercions`

Print the list of declared coercions in the current context.

Command: `Print Graph`

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

Command: `Print Coercion Paths` *class* *class*

Print the list of valid coercion paths between the two given classes.

8.2.8 Activating the Printing of Coercions

Flag: `Printing Coercions`

When on, this flag forces all the coercions to be printed. By default, coercions are not printed.

Table: `Printing Coercion` *qualid*

Specifies a set of qualids for which coercions are always displayed. Use the *Add @table* and *Remove @table* commands to update the set of qualids.

8.2.9 Classes as Records

We allow the definition of *Structures with Inheritance* (or classes as records) by extending the existing *Record* macro. Its new syntax is:

Variant: `Record` `>`? *ident* *binders*? : *sort* := *ident*? { *ident* :`>`? *term* +; }

The first identifier *ident* is the name of the defined record and *sort* is its type. The optional identifier after `:=` is the name of the constructor (it will be `Build_`*ident* if not given). The other identifiers are the names of the fields, and *term* are their respective types. If `:>` is used instead of `:` in the declaration of a field, then the name of this field is automatically declared as a coercion from the record name to the class of this field type. Note that the fields always verify the uniform inheritance condition. If the optional `>` is given before the record name, then the constructor name is automatically declared as a coercion from the class of the last field type to the record name (this may fail if the uniform inheritance condition is not satisfied).

Variant: `Structure` `>`? *ident* *binders*? : *sort* := *ident*? { *ident* :`>`? *term* +; }

This is a synonym of *Record*.

8.2.10 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.

8.2.11 Coercions and Modules

The coercions present in a module are activated only when the module is explicitly imported.

8.2.12 Examples

There are three situations:

Coercion at function application

`f a` is ill-typed where `f:forall x:A,B` and `a:A'`. If there is a coercion path between `A'` and `A`, then `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

```
Definition bool_in_nat (b:bool) := if b then 0 else 1.
    bool_in_nat is defined

Coercion bool_in_nat : bool >-> nat.
    bool_in_nat is now a coercion

Check (0 = true).
    0 = true
```

(continues on next page)

(continued from previous page)

```
         : Prop

Set Printing Coercions.
Check (0 = true).
    0 = bool_in_nat true
         : Prop

Unset Printing Coercions.
```

Warning: Note that `Check (true = 0)` would fail. This is ”normal” behavior 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.

```
Parameters (C : nat -> Set) (D : nat -> bool -> Set) (E : bool -> Set).
    C is declared
    D is declared
    E is declared

Parameter f : forall n:nat, C n -> D (S n) true.
    f is declared

Coercion f : C >-> D.
    f is now a coercion

Parameter g : forall (n:nat) (b:bool), D n b -> E b.
    g is declared

Coercion g : D >-> E.
    g is now a coercion

Parameter c : C 0.
    c is declared

Parameter T : E true -> nat.
    T is declared

Check (T c).
    T c
         : nat

Set Printing Coercions.
Check (T c).
    T (g 1 true (f 0 c))
         : nat

Unset Printing Coercions.
```

We give now an example using identity coercions.

```
Definition D' (b:bool) := D 1 b.
    D' is defined

Identity Coercion IdD'D : D' >-> D.
```

(continues on next page)

(continued from previous page)

```
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

    Arguments IdD'D _%bool_scope
    IdD'D is a coercion

Parameter d' : D' true.
    d' is declared

Check (T d').
    T d'
         : nat

Set Printing Coercions.
Check (T d').
    T (g 1 true d')
         : nat

Unset Printing Coercions.
```

In the case of functional arguments, we use the monotonic rule of sub-typing. To coerce `t : forall x : A, B` towards `forall x : A', B'`, we have to coerce `A'` towards `A` and `B` towards `B'`. An example is given below:

```
Parameters (A B : Set) (h : A -> B).
    A is declared
    B is declared
    h is declared

Coercion h : A >-> B.
    h is now a coercion

Parameter U : (A -> E true) -> nat.
    U is declared

Parameter t : B -> C 0.
    t is declared

Check (U t).
    U (fun x : A => t x)
         : nat

Set Printing Coercions.
Check (U t).
    U (fun x : A => g 1 true (f 0 (t (h x))))
         : nat

Unset Printing Coercions.
```

Remark the changes in the result following the modification of the previous example.

```
Parameter U' : (C 0 -> B) -> nat.
    U' is declared
```

(continues on next page)

(continued from previous page)

```
Parameter t' : E true -> A.
    t' is declared

Check (U' t').
    U' (fun x : C 0 => t' x)
         : nat

Set Printing Coercions.
Check (U' t').
    U' (fun x : C 0 => h (t' (g 1 true (f 0 x))))
         : nat

Unset Printing Coercions.
```

Coercion to a type

An assumption `x:A` when `A` is not a type, is ill-typed. It is replaced by `x:A'` where `A'` is the result of the application to `A` of the coercion path between the class of `A` and `Sortclass` if it exists. This case occurs in the abstraction `fun x:A => t`, universal quantification `forall x:A,B`, global variables and parameters of (co-)inductive definitions and functions. In `forall x:A,B`, such a coercion path may also be applied to `B` if necessary.

```
Parameter Graph : Type.
    Graph is declared

Parameter Node : Graph -> Type.
    Node is declared

Coercion Node : Graph >-> Sortclass.
    Node is now a coercion

Parameter G : Graph.
    G is declared

Parameter Arrows : G -> G -> Type.
    Arrows is declared

Check Arrows.
    Arrows
         : G -> G -> Type

Parameter fg : G -> G.
    fg is declared

Check fg.
    fg
         : G -> G

Set Printing Coercions.
Check fg.
    fg
         : Node G -> Node G

Unset Printing Coercions.
```

Coercion to a function

`f a` is ill-typed because `f:A` is not a function. The term `f` is replaced by the term obtained by applying to `f` the coercion path between `A` and `Funclass` if it exists.

```
Parameter bij : Set -> Set -> Set.
    bij is declared

Parameter ap : forall A B:Set, bij A B -> A -> B.
    ap is declared

Coercion ap : bij >-> Funclass.
    ap is now a coercion

Parameter b : bij nat nat.
    b is declared

Check (b 0).
    b 0
         : nat

Set Printing Coercions.
Check (b 0).
    ap nat nat b 0
         : nat

Unset Printing Coercions.
```

Let us see the resulting graph after all these examples.

```
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
```

8.3 Canonical Structures

Authors Assia Mahboubi and Enrico Tassi

This chapter explains the basics of canonical structures and how they can be used to overload notations and build a hierarchy of algebraic structures. The examples are taken from *[MT13]*. We invite the interested reader to refer to this paper for all the details that are omitted here for brevity. The interested reader shall also find in *[GZND11]* a detailed description of another, complementary, use of canonical structures: advanced proof search. This latter papers also presents many techniques one can employ to tune the inference of canonical structures.

8.3.1 Notation overloading

We build an infix notation == for a comparison predicate. Such notation will be overloaded, and its meaning will depend on the types of the terms that are compared.

```
Module EQ.
    Interactive Module EQ started

Record class (T : Type) := Class { cmp : T -> T -> Prop }.
    class is defined
    cmp is defined

Structure type := Pack { obj : Type; class_of : class obj }.
    type is defined
    obj is defined
    class_of is defined

Definition op (e : type) : obj e -> obj e -> Prop :=
    let 'Pack _ (Class _ the_cmp) := e in the_cmp.
    op is defined

Check op.
    op
         : forall e : type, obj e -> obj e -> Prop

Arguments op {e} x y : simpl never.
Arguments Class {T} cmp.
Module theory.
    Interactive Module theory started

Notation "x == y" := (op x y) (at level 70).
End theory.
    Module theory is defined

End EQ.
    Module EQ is defined
```

We use Coq modules as namespaces. This allows us to follow the same pattern and naming convention for the rest of the chapter. The base namespace contains the definitions of the algebraic structure. To keep the example small, the algebraic structure `EQ.type` we are defining is very simplistic, and characterizes terms on which a binary relation is defined, without requiring such relation to validate any property. The inner theory module contains the overloaded notation `==` and will eventually contain lemmas holding all the instances of the algebraic structure (in this case there are no lemmas).

Note that in practice the user may want to declare `EQ.obj` as a coercion, but we will not do that here.

The following line tests that, when we assume a type `e` that is in theEQ class, we can relate two of its objects with `==`.

```
Import EQ.theory.
Check forall (e : EQ.type) (a b : EQ.obj e), a == b.
    forall (e : EQ.type) (a b : EQ.obj e), a == b
         : Prop
```

Still, no concrete type is in the `EQ` class.

```
Fail Check 3 == 3.
    The command has indeed failed with message:
```

(continues on next page)

(continued from previous page)

```
    The term "3" has type "nat" while it is expected to have type "EQ.obj ?e".
```

We amend that by equipping `nat` with a comparison relation.

```
Definition nat_eq (x y : nat) := Nat.compare x y = Eq.
    nat_eq is defined

Definition nat_EQcl : EQ.class nat := EQ.Class nat_eq.
    nat_EQcl is defined

Canonical Structure nat_EQty : EQ.type := EQ.Pack nat nat_EQcl.
    nat_EQty is defined

Check 3 == 3.
    3 == 3
         : Prop

Eval compute in 3 == 4.
    = Lt = Eq
         : Prop
```

This last test shows that Coq is now not only able to type check `3 == 3`, but also that the infix relation was bound to the `nat_eq` relation. This relation is selected whenever `==` is used on terms of type nat. This can be read in the line declaring the canonical structure `nat_EQty`, where the first argument to `Pack` is the key and its second argument a group of canonical values associated to the key. In this case we associate to nat only one canonical value (since its class, `nat_EQcl` has just one member). The use of the projection `op` requires its argument to be in the class `EQ`, and uses such a member (function) to actually compare its arguments.

Similarly, we could equip any other type with a comparison relation, and use the `==` notation on terms of this type.

Derived Canonical Structures

We know how to use `==` on base types, like `nat`, `bool`, `Z`. Here we show how to deal with type constructors, i.e. how to make the following example work:

```
Fail Check forall (e : EQ.type) (a b : EQ.obj e), (a, b) == (a, b).
    The command has indeed failed with message:
    In environment
    e : EQ.type
    a : EQ.obj e
    b : EQ.obj e
    The term "(a, b)" has type "(EQ.obj e * EQ.obj e)%type"
    while it is expected to have type "EQ.obj ?e".
```

The error message is telling that Coq has no idea on how to compare pairs of objects. The following construction is telling Coq exactly how to do that.

```
Definition pair_eq (e1 e2 : EQ.type) (x y : EQ.obj e1 * EQ.obj e2) :=
  fst x == fst y /\ snd x == snd y.
    pair_eq is defined

Definition pair_EQcl e1 e2 := EQ.Class (pair_eq e1 e2).
    pair_EQcl is defined
```

(continues on next page)

(continued from previous page)

```
Canonical Structure pair_EQty (e1 e2 : EQ.type) : EQ.type :=
    EQ.Pack (EQ.obj e1 * EQ.obj e2) (pair_EQcl e1 e2).
    pair_EQty is defined

Check forall (e : EQ.type) (a b : EQ.obj e), (a, b) == (a, b).
    forall (e : EQ.type) (a b : EQ.obj e), (a, b) == (a, b)
         : Prop

Check forall n m : nat, (3, 4) == (n, m).
    forall n m : nat, (3, 4) == (n, m)
         : Prop
```

Thanks to the `pair_EQty` declaration, Coq is able to build a comparison relation for pairs whenever it is able to build a comparison relation for each component of the pair. The declaration associates to the key `*` (the type constructor of pairs) the canonical comparison relation `pair_eq` whenever the type constructor `*` is applied to two types being themselves in the `EQ` class.

8.3.2 Hierarchy of structures

To get to an interesting example we need another base class to be available. We choose the class of types that are equipped with an order relation, to which we associate the infix `<=` notation.

```
Module LE.
    Interactive Module LE started

Record class T := Class { cmp : T -> T -> Prop }.
    class is defined
    cmp is defined

Structure type := Pack { obj : Type; class_of : class obj }.
    type is defined
    obj is defined
    class_of is defined

Definition op (e : type) : obj e -> obj e -> Prop :=
    let 'Pack _ (Class _ f) := e in f.
    op is defined

Arguments op {_} x y : simpl never.
Arguments Class {T} cmp.
Module theory.
    Interactive Module theory started

Notation "x <= y" := (op x y) (at level 70).
End theory.
    Module theory is defined

End LE.
    Module LE is defined
```

As before we register a canonical `LE` class for `nat`.

```
Import LE.theory.
Definition nat_le x y := Nat.compare x y <> Gt.
```

(continues on next page)

(continued from previous page)

```
    nat_le is defined

Definition nat_LEcl : LE.class nat := LE.Class nat_le.
    nat_LEcl is defined

Canonical Structure nat_LEty : LE.type := LE.Pack nat nat_LEcl.
    nat_LEty is defined
```

And we enable Coq to relate pair of terms with `<=`.

```
Definition pair_le e1 e2 (x y : LE.obj e1 * LE.obj e2) :=
   fst x <= fst y /\ snd x <= snd y.
    pair_le is defined

Definition pair_LEcl e1 e2 := LE.Class (pair_le e1 e2).
    pair_LEcl is defined

Canonical Structure pair_LEty (e1 e2 : LE.type) : LE.type :=
   LE.Pack (LE.obj e1 * LE.obj e2) (pair_LEcl e1 e2).
    pair_LEty is defined

Check (3,4,5) <= (3,4,5).
    (3, 4, 5) <= (3, 4, 5)
         : Prop
```

At the current stage we can use `==` and `<=` on concrete types, like tuples of natural numbers, but we can't develop an algebraic theory over the types that are equipped with both relations.

```
Check 2 <= 3 /\ 2 == 2.
    2 <= 3 /\ 2 == 2
         : Prop

Fail Check forall (e : EQ.type) (x y : EQ.obj e), x <= y -> y <= x -> x == y.
    The command has indeed failed with message:
    In environment
    e : EQ.type
    x : EQ.obj e
    y : EQ.obj e
    The term "x" has type "EQ.obj e" while it is expected to have type
     "LE.obj ?e".

Fail Check forall (e : LE.type) (x y : LE.obj e), x <= y -> y <= x -> x == y.
    The command has indeed failed with message:
    In environment
    e : LE.type
    x : LE.obj e
    y : LE.obj e
    The term "x" has type "LE.obj e" while it is expected to have type
     "EQ.obj ?e".
```

We need to define a new class that inherits from both `EQ` and `LE`.

```
Module LEQ.
    Interactive Module LEQ started

Record mixin (e : EQ.type) (le : EQ.obj e -> EQ.obj e -> Prop) :=
    Mixin { compat : forall x y : EQ.obj e, le x y /\ le y x <-> x == y }.
```

(continues on next page)

(continued from previous page)

```
    mixin is defined
    compat is defined

Record class T := Class {
                      EQ_class : EQ.class T;
                      LE_class : LE.class T;
                      extra : mixin (EQ.Pack T EQ_class) (LE.cmp T LE_class) }.
    class is defined
    EQ_class is defined
    LE_class is defined
    extra is defined

Structure type := _Pack { obj : Type; #[canonical(false)] class_of : class obj }.
    type is defined
    obj is defined
    class_of is defined

Arguments Mixin {e le} _.
Arguments Class {T} _ _ _.
```

The mixin component of the LEQ class contains all the extra content we are adding to EQ and LE. In particular it contains the requirement that the two relations we are combining are compatible.

The class_of projection of the type structure is annotated as *not canonical*; it plays no role in the search for instances.

Unfortunately there is still an obstacle to developing the algebraic theory of this new class.

```
Module theory.
    Interactive Module theory started

Fail Check forall (le : type) (n m : obj le), n <= m -> n <= m -> n == m.
    The command has indeed failed with message:
    In environment
    le : type
    n : obj le
    m : obj le
    The term "n" has type "obj le" while it is expected to have type "LE.obj ?e".
```

The problem is that the two classes LE and LEQ are not yet related by a subclass relation. In other words Coq does not see that an object of the LEQ class is also an object of the LE class.

The following two constructions tell Coq how to canonically build the LE.type and EQ.type structure given an LEQ.type structure on the same type.

```
Definition to_EQ (e : type) : EQ.type :=
   EQ.Pack (obj e) (EQ_class _ (class_of e)).
    to_EQ is defined

Canonical Structure to_EQ.
Definition to_LE (e : type) : LE.type :=
   LE.Pack (obj e) (LE_class _ (class_of e)).
    to_LE is defined

Canonical Structure to_LE.
```

We can now formulate out first theorem on the objects of the LEQ structure.

```
   Lemma lele_eq (e : type) (x y : obj e) : x <= y -> y <= x -> x == y.
    1 subgoal

      e : type
      x, y : obj e
      ============================
      x <= y -> y <= x -> x == y

now intros; apply (compat _ _ (extra _ (class_of e)) x y); split.
    No more subgoals.

Qed.
Arguments lele_eq {e} x y _ _.
End theory.
    Module theory is defined

End LEQ.
    Module LEQ is defined

Import LEQ.theory.
Check lele_eq.
    lele_eq
         : forall x y : LEQ.obj ?e, x <= y -> y <= x -> x == y
    where
    ?e : [ |- LEQ.type]
```

Of course one would like to apply results proved in the algebraic setting to any concrete instate of the algebraic structure.

```
Example test_algebraic (n m : nat) : n <= m -> m <= n -> n == m.
    1 subgoal

      n, m : nat
      ============================
      n <= m -> m <= n -> n == m

Fail apply (lele_eq n m).
    The command has indeed failed with message:
    In environment
    n, m : nat
    The term "n" has type "nat" while it is expected to have type "LEQ.obj ?e".

Abort.
Example test_algebraic2 (l1 l2 : LEQ.type) (n m : LEQ.obj l1 * LEQ.obj l2) :
     n <= m -> m <= n -> n == m.
    1 subgoal

      l1, l2 : LEQ.type
      n, m : LEQ.obj l1 * LEQ.obj l2
      ============================
      n <= m -> m <= n -> n == m

Fail apply (lele_eq n m).
    The command has indeed failed with message:
    In environment
    l1, l2 : LEQ.type
    n, m : LEQ.obj l1 * LEQ.obj l2
```

(continues on next page)

(continued from previous page)

```
    The term "n" has type "(LEQ.obj l1 * LEQ.obj l2)%type"
    while it is expected to have type "LEQ.obj ?e".

Abort.
```

Again one has to tell Coq that the type `nat` is in the `LEQ` class, and how the type constructor `*` interacts with the `LEQ` class. In the following proofs are omitted for brevity.

```
Lemma nat_LEQ_compat (n m : nat) : n <= m /\ m <= n <-> n == m.
    1 subgoal

      n, m : nat
      ============================
      n <= m /\ m <= n <-> n == m

Admitted.
    nat_LEQ_compat is declared

Definition nat_LEQmx := LEQ.Mixin nat_LEQ_compat.
    nat_LEQmx is defined

Lemma pair_LEQ_compat (l1 l2 : LEQ.type) (n m : LEQ.obj l1 * LEQ.obj l2) :
   n <= m /\ m <= n <-> n == m.
    1 subgoal

      l1, l2 : LEQ.type
      n, m : LEQ.obj l1 * LEQ.obj l2
      ============================
      n <= m /\ m <= n <-> n == m

Admitted.
    pair_LEQ_compat is declared

Definition pair_LEQmx l1 l2 := LEQ.Mixin (pair_LEQ_compat l1 l2).
    pair_LEQmx is defined
```

The following script registers an `LEQ` class for `nat` and for the type constructor `*`. It also tests that they work as expected.

Unfortunately, these declarations are very verbose. In the following subsection we show how to make them more compact.

```
Module Add_instance_attempt.
    Interactive Module Add_instance_attempt started

Canonical Structure nat_LEQty : LEQ.type :=
    LEQ._Pack nat (LEQ.Class nat_EQcl nat_LEcl nat_LEQmx).
    nat_LEQty is defined

Canonical Structure pair_LEQty (l1 l2 : LEQ.type) : LEQ.type :=
    LEQ._Pack (LEQ.obj l1 * LEQ.obj l2)
      (LEQ.Class
         (EQ.class_of (pair_EQty (to_EQ l1) (to_EQ l2)))
         (LE.class_of (pair_LEty (to_LE l1) (to_LE l2)))
         (pair_LEQmx l1 l2)).
    pair_LEQty is defined
```

(continues on next page)

(continued from previous page)

```
Example test_algebraic (n m : nat) : n <= m -> m <= n -> n == m.
    1 subgoal

      n, m : nat
      ============================
      n <= m -> m <= n -> n == m

now apply (lele_eq n m).
    No more subgoals.

Qed.
Example test_algebraic2 (n m : nat * nat) : n <= m -> m <= n -> n == m.
    1 subgoal

      n, m : nat * nat
      ============================
      n <= m -> m <= n -> n == m

now apply (lele_eq n m).
    No more subgoals.

Qed.
End Add_instance_attempt.
    Module Add_instance_attempt is defined
```

Note that no direct proof of `n <= m -> m <= n -> n == m` is provided by the user for `n` and m of type `nat * nat`. What the user provides is a proof of this statement for `n` and `m` of type `nat` and a proof that the pair constructor preserves this property. The combination of these two facts is a simple form of proof search that Coq performs automatically while inferring canonical structures.

Compact declaration of Canonical Structures

We need some infrastructure for that.

```
Require Import Strings.String.
    [Loading ML file newring_plugin.cmxs ... done]

Module infrastructure.
    Interactive Module infrastructure started

Inductive phantom {T : Type} (t : T) : Type := Phantom.
    phantom is defined
    phantom_rect is defined
    phantom_ind is defined
    phantom_rec is defined
    phantom_sind is defined

Definition unify {T1 T2} (t1 : T1) (t2 : T2) (s : option string) :=
    phantom t1 -> phantom t2.
    unify is defined

Definition id {T} {t : T} (x : phantom t) := x.
    id is defined

Notation "[find v | t1 ~ t2 ] p" := (fun v (_ : unify t1 t2 None) => p)
```

(continues on next page)

(continued from previous page)

```
    (at level 50, v ident, only parsing).
Notation "[find v | t1 ~ t2 | s ] p" := (fun v (_ : unify t1 t2 (Some s)) => p)
    (at level 50, v ident, only parsing).
Notation "'Error : t : s" := (unify _ t (Some s))
    (at level 50, format "''Error' : t : s").
Open Scope string_scope.
End infrastructure.
    Module infrastructure is defined
```

To explain the notation `[find v | t1 ~ t2]` let us pick one of its instances: `[find e | EQ.obj e ~ T | "is not an EQ.type" ]`. It should be read as: "find a class e such that its objects have type T or fail with message "T is not an EQ.type"".

The other utilities are used to ask Coq to solve a specific unification problem, that will in turn require the inference of some canonical structures. They are explained in more details in *[MT13]*.

We now have all we need to create a compact "packager" to declare instances of the `LEQ` class.

```
Import infrastructure.
Definition packager T e0 le0 (m0 : LEQ.mixin e0 le0) :=
  [find e | EQ.obj e ~ T | "is not an EQ.type" ]
  [find o | LE.obj o ~ T | "is not an LE.type" ]
  [find ce | EQ.class_of e ~ ce ]
  [find co | LE.class_of o ~ co ]
  [find m | m ~ m0 | "is not the right mixin" ]
  LEQ._Pack T (LEQ.Class ce co m).
    packager is defined

Notation Pack T m := (packager T _ _ m _ id _ id _ id _ id _ id).
```

The object `Pack` takes a type `T` (the key) and a mixin `m`. It infers all the other pieces of the class `LEQ` and declares them as canonical values associated to the `T` key. All in all, the only new piece of information we add in the `LEQ` class is the mixin, all the rest is already canonical for `T` and hence can be inferred by Coq.

`Pack` is a notation, hence it is not type checked at the time of its declaration. It will be type checked when it is used, an in that case `T` is going to be a concrete type. The odd arguments `_` and `id` we pass to the packager represent respectively the classes to be inferred (like `e`, `o`, etc) and a token (`id`) to force their inference. Again, for all the details the reader can refer to *[MT13]*.

The declaration of canonical instances can now be way more compact:

```
Canonical Structure nat_LEQty := Eval hnf in Pack nat nat_LEQmx.
    nat_LEQty is defined

Canonical Structure pair_LEQty (l1 l2 : LEQ.type) :=
   Eval hnf in Pack (LEQ.obj l1 * LEQ.obj l2) (pair_LEQmx l1 l2).
    pair_LEQty is defined
```

Error messages are also quite intelligible (if one skips to the end of the message).

```
Fail Canonical Structure err := Eval hnf in Pack bool nat_LEQmx.
    The command has indeed failed with message:
    The term "id" has type "phantom (EQ.obj ?e) -> phantom (EQ.obj ?e)"
    while it is expected to have type "'Error:bool:"is not an EQ.type"".
```

8.4 Typeclasses

This chapter presents a quick reference of the commands related to type classes. For an actual introduction to typeclasses, there is a description of the system [SO08] and the literature on type classes in Haskell which also applies.

8.4.1 Class and Instance declarations

The syntax for class and instance declarations is the same as the record syntax of Coq:

```
Class classname (p1 : t1)   (pn : tn) [: sort] := { f1 : u1 ;   ; fm : um }.

Instance instancename q1   qm : classname p1   pn := { f1 := t1 ;   ; fm := tm }.
```

The `pi : ti` variables are called the *parameters* of the class and the `fi : ti` are called the *methods*. Each class definition gives rise to a corresponding record declaration and each instance is a regular definition whose name is given by `instancename` and type is an instantiation of the record type.

We'll use the following example class in the rest of the chapter:

```
Class EqDec (A : Type) :=
  { eqb : A -> A -> bool ;
    eqb_leibniz : forall x y, eqb x y = true -> x = y }.
```

This class implements a boolean equality test which is compatible with Leibniz equality on some type. An example implementation is:

```
Instance unit_EqDec : EqDec unit :=
  { eqb x y := true ;
    eqb_leibniz x y H :=
      match x, y return x = y with
      | tt, tt => eq_refl tt
      end }.
```

Using the attribute `refine`, if the term is not sufficient to finish the definition (e.g. due to a missing field or non-inferable hole) it must be finished in proof mode. If it is sufficient a trivial proof mode with no open goals is started.

```
#[refine] Instance unit_EqDec' : EqDec unit := { eqb x y := true }.
Proof.
intros [] [];reflexivity.
Defined.
```

Note that if you finish the proof with *`Qed`* the entire instance will be opaque, including the fields given in the initial term.

Alternatively, in *`Program Mode`* if one does not give all the members in the Instance declaration, Coq generates obligations for the remaining fields, e.g.:

```
Require Import Program.Tactics.
Program Instance eq_bool : EqDec bool :=
  { eqb x y := if x then y else negb y }.


Next Obligation.
    1 subgoal
```

(continues on next page)

(continued from previous page)

```
      x, y : bool
      H : (if x then y else negb y) = true
      ============================
      x = y

destruct x ; destruct y ; (discriminate || reflexivity).
    No more subgoals.

Defined.
```

One has to take care that the transparency of every field is determined by the transparency of the *Instance* proof. One can use alternatively the *Program Instance* variant which has richer facilities for dealing with obligations.

8.4.2 Binding classes

Once a typeclass is declared, one can use it in class binders:

```
Definition neqb {A} {eqa : EqDec A} (x y : A) := negb (eqb x y).
    neqb is defined
```

When one calls a class method, a constraint is generated that is satisfied only in contexts where the appropriate instances can be found. In the example above, a constraint `EqDec A` is generated and satisfied by `eqa : EqDec A`. In case no satisfying constraint can be found, an error is raised:

```
Fail Definition neqb' (A : Type) (x y : A) := negb (eqb x y).
    The command has indeed failed with message:
    Unable to satisfy the following constraints:
    In environment:
    A : Type
    x, y : A

    ?EqDec : "EqDec A"
```

The algorithm used to solve constraints is a variant of the *eauto* tactic that does proof search with a set of lemmas (the instances). It will use local hypotheses as well as declared lemmas in the `typeclass_instances` database. Hence the example can also be written:

```
Definition neqb' A (eqa : EqDec A) (x y : A) := negb (eqb x y).
    neqb' is defined
```

However, the generalizing binders should be used instead as they have particular support for typeclasses:

- They automatically set the maximally implicit status for typeclass arguments, making derived functions as easy to use as class methods. In the example above, `A` and `eqa` should be set maximally implicit.
- They support implicit quantification on partially applied type classes (*Implicit generalization*). Any argument not given as part of a typeclass binder will be automatically generalized.
- They also support implicit quantification on *Superclasses*.

Following the previous example, one can write:

```
Generalizable Variables A B C.
Definition neqb_implicit `{eqa : EqDec A} (x y : A) := negb (eqb x y).
    neqb_implicit is defined
```

Here A is implicitly generalized, and the resulting function is equivalent to the one above.

8.4.3 Parameterized Instances

One can declare parameterized instances as in Haskell simply by giving the constraints as a binding context before the instance, e.g.:

```
Program Instance prod_eqb `(EA : EqDec A, EB : EqDec B) : EqDec (A * B) :=
  { eqb x y := match x, y with
               | (la, ra), (lb, rb) => andb (eqb la lb) (eqb ra rb)
               end }.
```

These instances are used just as well as lemmas in the instance hint database.

8.4.4 Sections and contexts

To ease developments parameterized by many instances, one can use the *Context* command to introduce these parameters into section contexts, it works similarly to the command *Variable*, except it accepts any binding context as an argument, so variables can be implicit, and *Implicit generalization* can be used. For example:

```
Section EqDec_defs.
Context `{EA : EqDec A}.
    A is declared
    EA is declared


Global Program Instance option_eqb : EqDec (option A) :=
  { eqb x y := match x, y with
         | Some x, Some y => eqb x y
         | None, None => true
         | _, _ => false
         end }.
Admit Obligations.


End EqDec_defs.
About option_eqb.
    option_eqb : forall A : Type, EqDec A -> EqDec (option A)

    option_eqb is not universe polymorphic
    Arguments option_eqb {A}%type_scope {EA}
    option_eqb is transparent
    Expands to: Constant Top.option_eqb
```

Here the *Global* modifier redeclares the instance at the end of the section, once it has been generalized by the context variables it uses.

See also:

Section *Section mechanism*

8.4.5 Building hierarchies

Superclasses

One can also parameterize classes by other classes, generating a hierarchy of classes and superclasses. In the same way, we give the superclasses as a binding context:

```
Class Ord `(E : EqDec A) := { le : A -> A -> bool }.
    Ord is defined
    le is defined
```

Contrary to Haskell, we have no special syntax for superclasses, but this declaration is equivalent to:

```
Class `(E : EqDec A) => Ord A :=
  { le : A -> A -> bool }.
```

This declaration means that any instance of the Ord class must have an instance of EqDec. The parameters of the subclass contain at least all the parameters of its superclasses in their order of appearance (here A is the only one). As we have seen, Ord is encoded as a record type with two parameters: a type A and an E of type EqDec A. However, one can still use it as if it had a single parameter inside generalizing binders: the generalization of superclasses will be done automatically.

```
Definition le_eqb `{Ord A} (x y : A) := andb (le x y) (le y x).
    le_eqb is defined
```

In some cases, to be able to specify sharing of structures, one may want to give explicitly the superclasses. It is is possible to do it directly in regular binders, and using the ! modifier in class binders. For example:

```
Definition lt `{eqa : EqDec A, ! Ord eqa} (x y : A) := andb (le x y) (neqb x y).
    lt is defined
```

The ! modifier switches the way a binder is parsed back to the regular interpretation of Coq. In particular, it uses the implicit arguments mechanism if available, as shown in the example.

Substructures

Substructures are components of a class which are instances of a class themselves. They often arise when using classes for logical properties, e.g.:

```
Class Reflexive (A : Type) (R : relation A) :=
  reflexivity : forall x, R x x.
Class Transitive (A : Type) (R : relation A) :=
  transitivity : forall x y z, R x y -> R y z -> R x z.
```

This declares singleton classes for reflexive and transitive relations, (see the *singleton class* variant for an explanation). These may be used as parts of other classes:

```
Class PreOrder (A : Type) (R : relation A) :=
  { PreOrder_Reflexive :> Reflexive A R ;
    PreOrder_Transitive :> Transitive A R }.
    PreOrder is defined
    PreOrder_Reflexive is defined
    PreOrder_Transitive is defined
```

The syntax :> indicates that each PreOrder can be seen as a Reflexive relation. So each time a reflexive relation is needed, a preorder can be used instead. This is very similar to the coercion mechanism of Structure declarations. The implementation simply declares each projection as an instance.

Warning: `Ignored instance declaration for "ident": "term" is not a class`

Using this `:>` syntax with a right-hand-side that is not itself a Class has no effect (apart from emitting this warning). In particular, is does not declare a coercion.

One can also declare existing objects or structure projections using the Existing Instance command to achieve the same effect.

8.4.6 Summary of the commands

Command: `Class ident binders? : sort? := ident? { ident :>? term+; }`

The *Class* command is used to declare a typeclass with parameters *binders* and fields the declared record fields.

Variant: `Class ident binders? : sort? := ident : term`

This variant declares a *singleton* class with a single method. This singleton class is a so-called definitional class, represented simply as a definition `ident binders := term` and whose instances are themselves objects of this type. Definitional classes are not wrapped inside records, and the trivial projection of an instance of such a class is convertible to the instance itself. This can be useful to make instances of existing objects easily and to reduce proof size by not inserting useless projections. The class constant itself is declared rigid during resolution so that the class abstraction is maintained.

Variant: `Existing Class ident`

This variant declares a class a posteriori from a constant or inductive definition. No methods or instances are defined.

Warning: `ident is already declared as a typeclass`

This command has no effect when used on a typeclass.

Command: `Instance ident binders? : term0 term+ | num? := { field_def*; }`

This command is used to declare a typeclass instance named *ident* of the class $term_0$ with parameters *term* and fields defined by *field_def*, where each field must be a declared field of the class.

An arbitrary context of *binders* can be put after the name of the instance and before the colon to declare a parameterized instance. An optional priority can be declared, 0 being the highest priority as for *auto* hints. If the priority *num* is not specified, it defaults to the number of non-dependent binders of the instance.

Variant: `Instance ident binders? : forall binders?, term0 term+ | num? := term`

This syntax is used for declaration of singleton class instances or for directly giving an explicit term of type `forall binders, term0 term+`. One need not even mention the unique field name for singleton classes.

Variant: `Global Instance`

One can use the *Global* modifier on instances declared in a section so that their generalization is automatically redeclared after the section is closed.

Variant: `Program Instance`

Switches the type checking to `Program` (chapter *Program*) and uses the obligation mechanism to manage missing fields.

Variant: `Declare Instance`

In a *Module Type*, this command states that a corresponding concrete instance should exist in any implementation of this *Module Type*. This is similar to the distinction between *Parameter* vs. *Definition*, or between *Declare Module* and *Module*.

Besides the *Class* and *Instance* vernacular commands, there are a few other commands related to typeclasses.

Command: `Existing Instance` *ident*+ `| num`?

This command adds an arbitrary list of constants whose type ends with an applied typeclass to the instance database with an optional priority *num*. It can be used for redeclaring instances at the end of sections, or declaring structure projections as instances. This is equivalent to `Hint Resolve ident : typeclass_instances`, except it registers instances for *Print Instances*.

`typeclasses eauto`

This tactic uses a different resolution engine than *eauto* and *auto*. The main differences are the following:

- Contrary to *eauto* and *auto*, the resolution is done entirely in the new proof engine (as of Coq 8.6), meaning that backtracking is available among dependent subgoals, and shelving goals is supported. `typeclasses eauto` is a multi-goal tactic. It analyses the dependencies between subgoals to avoid backtracking on subgoals that are entirely independent.
- When called with no arguments, `typeclasses eauto` uses the `typeclass_instances` database by default (instead of core). Dependent subgoals are automatically shelved, and shelved goals can remain after resolution ends (following the behavior of Coq 8.5).

Note: As of Coq 8.6, `all:once (typeclasses eauto)` faithfully mimics what happens during typeclass resolution when it is called during refinement/type inference, except that *only* declared class subgoals are considered at the start of resolution during type inference, while `all` can select non-class subgoals as well. It might move to `all:typeclasses eauto` in future versions when the refinement engine will be able to backtrack.

- When called with specific databases (e.g. with), `typeclasses eauto` allows shelved goals to remain at any point during search and treat typeclass goals like any other.
- The transparency information of databases is used consistently for all hints declared in them. It is always used when calling the unifier. When considering local hypotheses, we use the transparent state of the first hint database given. Using an empty database (created with *Create HintDb* for example) with unfoldable variables and constants as the first argument of `typeclasses eauto` hence makes resolution with the local hypotheses use full conversion during unification.
- When considering local hypotheses, we use the union of all the modes declared in the given databases.

Variant: `typeclasses eauto` *num*

Warning: The semantics for the limit *num* is different than for auto. By default, if no limit is given, the search is unbounded. Contrary to *auto*, introduction steps are counted, which might result in larger limits being necessary when searching with `typeclasses eauto` than with *auto*.

Variant: `typeclasses eauto with` *ident*+

This variant runs resolution with the given hint databases. It treats typeclass subgoals the same as other subgoals (no shelving of non-typeclass goals in particular).

`autoapply` *term* `with` *ident*

The tactic `autoapply` applies a term using the transparency information of the hint database ident, and does *no* typeclass resolution. This can be used in *Hint Extern*'s for typeclass instances (in the hint database `typeclass_instances`) to allow backtracking on the typeclass subgoals created by the lemma application, rather than doing typeclass resolution locally at the hint application time.

Typeclasses Transparent, Typeclasses Opaque

Command: `Typeclasses Transparent` *ident*+

This command makes the identifiers transparent during typeclass resolution. Shortcut for `Hint Transparent` *ident*+ `: typeclass_instances`.

Command: `Typeclasses Opaque` *ident*+

Make the identifiers opaque for typeclass search. Shortcut for `Hint Opaque` *ident*+ `: typeclass_instances`.

It is useful when some constants prevent some unifications and make resolution fail. It is also useful to declare constants which should never be unfolded during proof-search, like fixpoints or anything which does not look like an abbreviation. This can additionally speed up proof search as the typeclass map can be indexed by such rigid constants (see *The hints databases for auto and eauto*).

By default, all constants and local variables are considered transparent. One should take care not to make opaque any constant that is used to abbreviate a type, like:

```
Definition relation A := A -> A -> Prop.
```

Settings

Flag: `Typeclasses Dependency Order`

This flag (off by default) respects the dependency order between subgoals, meaning that subgoals on which other subgoals depend come first, while the non-dependent subgoals were put before the dependent ones previously (Coq 8.5 and below). This can result in quite different performance behaviors of proof search.

Flag: `Typeclasses Filtered Unification`

This flag, available since Coq 8.6 and off by default, switches the hint application procedure to a filter-then-unify strategy. To apply a hint, we first check that the goal *matches* syntactically the inferred or specified pattern of the hint, and only then try to *unify* the goal with the conclusion of the hint. This can drastically improve performance by calling unification less often, matching syntactic patterns being very quick. This also provides more control on the triggering of instances. For example, forcing a constant to explicitly appear in the pattern will make it never apply on a goal where there is a hole in that place.

Flag: `Typeclasses Limit Intros`

This flag (on by default) controls the ability to apply hints while avoiding (functional) eta-expansions in the generated proof term. It does so by allowing hints that conclude in a product to apply to a goal with a matching product directly, avoiding an introduction.

> **Warning:** This can be expensive as it requires rebuilding hint clauses dynamically, and does not benefit from the invertibility status of the product introduction rule, resulting in potentially more expensive proof-search (i.e. more useless backtracking).

Flag: `Typeclass Resolution For Conversion`

This flag (on by default) controls the use of typeclass resolution when a unification problem cannot be solved during elaboration/type inference. With this flag on, when a unification fails, typeclass resolution is tried before launching unification once again.

Flag: `Typeclasses Strict Resolution`

Typeclass declarations introduced when this flag is set have a stricter resolution behavior (the flag is off by default). When looking for unifications of a goal with an instance of this class, we "freeze" all the existentials appearing in the goals, meaning that they are considered rigid during unification and cannot be instantiated.

Flag: `Typeclasses Unique Solutions`

When a typeclass resolution is launched we ensure that it has a single solution or fail. This ensures that the resolution is canonical, but can make proof search much more expensive.

Flag: `Typeclasses Unique Instances`

Typeclass declarations introduced when this flag is set have a more efficient resolution behavior (the flag is off by default). When a solution to the typeclass goal of this class is found, we never backtrack on it, assuming that it is canonical.

Flag: `Typeclasses Iterative Deepening`

When this flag is set, the proof search strategy is breadth-first search. Otherwise, the search strategy is depth-first search. The default is off. *`Typeclasses eauto`* is another way to set this flag.

Option: `Typeclasses Depth` *`num`*

Sets the maximum proof search depth. The default is unbounded. *`Typeclasses eauto`* is another way to set this option.

Flag: `Typeclasses Debug`

Controls whether typeclass resolution steps are shown during search. Setting this flag also sets *`Typeclasses Debug Verbosity`* to 1. *`Typeclasses eauto`* is another way to set this flag.

Option: `Typeclasses Debug Verbosity` *`num`*

Determines how much information is shown for typeclass resolution steps during search. 1 is the default level. 2 shows additional information such as tried tactics and shelving of goals. Setting this option to 1 or 2 turns on the *`Typeclasses Debug`* flag; setting this option to 0 turns that flag off.

Flag: `Typeclasses Axioms Are Instances`

Deprecated since version 8.10.

This flag (off by default since 8.8) automatically declares axioms whose type is a typeclass at declaration time as instances of that class.

Typeclasses eauto :=

Command: `Typeclasses eauto := debug`? `(dfs) | (bfs)`? *`num`*

This command allows more global customization of the typeclass resolution tactic. The semantics of the options are:

- `debug` This sets the debug mode. In debug mode, the trace of successfully applied tactics is printed. The debug mode can also be set with *`Typeclasses Debug`*.
- `(dfs)`, `(bfs)` This sets the search strategy to depth-first search (the default) or breadth-first search. The search strategy can also be set with *`Typeclasses Iterative Deepening`*.
- *`num`* This sets the depth limit of the search. The depth limit can also be set with *`Typeclasses Depth`*.

8.5 Omega: a solver for quantifier-free problems in Presburger Arithmetic

Author Pierre Crégut

Warning: The *omega* tactic is about to be deprecated in favor of the *lia* tactic. The goal is to consolidate the arithmetic solving capabilities of Coq into a single engine; moreover, *lia* is in general more powerful than *omega* (it is a complete Presburger arithmetic solver while *omega* was known to be incomplete).

Work is in progress to make sure that there are no regressions (including no performance regression) when switching from *omega* to *lia* in existing projects. However, we already recommend using *lia* in new or refactored proof scripts. We also ask that you report (in our bug tracker[313]) any issue you encounter, especially if the issue was not present in *omega*.

Note that replacing *omega* with *lia* can break non-robust proof scripts which rely on incompleteness bugs of *omega* (e.g. using the pattern `; try omega`).

8.5.1 Description of `omega`

`omega`

omega is a tactic for solving goals in Presburger arithmetic, i.e. for proving formulas made of equations and inequalities over the type `nat` of natural numbers or the type `Z` of binary-encoded integers. 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.

8.5.2 Arithmetical goals recognized by `omega`

omega applies only to quantifier-free formulas built from the connectives:

```
/\  \/  ~  ->
```

on atomic formulas. Atomic formulas are built from the predicates:

```
=  <  <=  >  >=
```

on `nat` or `Z`. In expressions of type `nat`, *omega* recognizes:

```
+  -  *  S  O  pred
```

and in expressions of type `Z`, *omega* recognizes numeral constants and:

```
+  -  *  Z.succ Z.pred
```

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`.

[313] https://github.com/coq/coq/issues

8.5.3 Messages from `omega`

When *omega* does not solve the goal, one of the following errors is generated:

Error: **`omega can't solve this system.`**

This may happen if your goal is not quantifier-free (if it is universally quantified, try *intros* first; if it contains existentials quantifiers too, *omega* is not strong enough to solve your goal). This may happen also if your goal contains arithmetical operators not recognized by *omega*. Finally, your goal may be simply not true!

Error: **`omega: Not a quantifier-free goal.`**

If your goal is universally quantified, you should first apply *intro* as many times as needed.

Error: **`omega: Unrecognized predicate or connective:`** ***`ident`*****`.`**

Error: **`omega: Unrecognized atomic proposition: ...`**

Error: **`omega: Can't solve a goal with proposition variables.`**

Error: **`omega: Unrecognized proposition.`**

Error: **`omega: Can't solve a goal with non-linear products.`**

Error: **`omega: Can't solve a goal with equality on type ...`**

8.5.4 Using `omega`

The `omega` tactic does not belong to the core system. It should be loaded by

```
Require Import Omega.
```

Example

```
Require Import Omega.
Open Scope Z_scope.
Goal forall m n:Z, 1 + 2 * m <> 2 * n.
    1 subgoal

      ============================
      forall m n : Z, 1 + 2 * m <> 2 * n

intros; omega.
    No more subgoals.

Abort.
Goal forall z:Z, z > 0 -> 2 * z + 1 > z.
    1 subgoal

      ============================
      forall z : Z, z > 0 -> 2 * z + 1 > z

intro; omega.
    No more subgoals.

Abort.
```

8.5.5 Options

Flag: `Stable Omega`

Deprecated since version 8.5.

This deprecated flag (on by default) is for compatibility with Coq pre 8.5. It resets internal name counters to make executions of *omega* independent.

Flag: `Omega UseLocalDefs`

This flag (on by default) allows *omega* to use the bodies of local variables.

Flag: `Omega System`

This flag (off by default) activate the printing of debug information

Flag: `Omega Action`

This flag (off by default) activate the printing of debug information

8.5.6 Technical data

Overview of the tactic

- The goal is negated twice and the first negation is introduced as a hypothesis.
- Hypotheses are decomposed in simple equations or inequalities. Multiple goals may result from this phase.
- Equations and inequalities over `nat` are translated over `Z`, multiple goals may result from the translation of subtraction.
- Equations and inequalities are normalized.
- Goals are solved by the OMEGA decision procedure.
- The script of the solution is replayed.

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 *[Pug92]* Here is an overview, refer to the original paper for more information.

- Equations and inequalities 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 inequality cuts the Euclidean space in half.
- Inequalities are solved by projecting on the hyperspace defined by cancelling one of the variables. They are partitioned according to the sign of the coefficient of the eliminated variable. Pairs of inequalities from different classes define a new edge in the projection.
- Redundant inequalities 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 are not implemented, so the decision procedure is only partial.

8.5.7 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 https://coq.inria.fr/bugs/.

8.6 Micromega: tactics for solving arithmetic goals over ordered rings

Authors Frédéric Besson and Evgeny Makarov

8.6.1 Short description of the tactics

The Psatz module (`Require Import Psatz.`) gives access to several tactics for solving arithmetic goals over $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{Z}$ but also `nat` and `N`. It also possible to get the tactics for integers by a `Require Import Lia`, rationals `Require Import Lqa` and reals `Require Import Lra`.

- *lia* is a decision procedure for linear integer arithmetic;
- *nia* is an incomplete proof procedure for integer non-linear arithmetic;
- *lra* is a decision procedure for linear (real or rational) arithmetic;
- *nra* is an incomplete proof procedure for non-linear (real or rational) arithmetic;
- *psatz* `D n` where `D` is $\mathbb{Z}$ or $\mathbb{Q}$ or $\mathbb{R}$, and `n` is an optional integer limiting the proof search depth, is an incomplete proof procedure for non-linear arithmetic. It is based on John Harrison's HOL Light driver to the external prover `csdp`[314]. Note that the `csdp` driver is generating a *proof cache* which makes it possible to rerun scripts even without `csdp`.

Flag: `Simplex`
This flag (set by default) instructs the decision procedures to use the Simplex method for solving linear goals. If it is not set, the decision procedures are using Fourier elimination.

Flag: `Lia Cache`
This flag (set by default) instructs *lia* to cache its results in the file `.lia.cache`

Flag: `Nia Cache`
This flag (set by default) instructs *nia* to cache its results in the file `.nia.cache`

Flag: `Nra Cache`
This flag (set by default) instructs *nra* to cache its results in the file `.nra.cache`

The tactics solve propositional formulas parameterized by atomic arithmetic expressions interpreted over a domain $D \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$. The syntax of the formulas is the following:

```
F  ::=   A   P   True   False   F   F   F   F   F   F   F → F   ¬ F
A  ::=   p = p   p > p   p < p   p   p   p   p
p  ::=   c   x   -p   p - p   p + p   p × p   p ^ n
```

where c is a numeric constant, $x \in D$ is a numeric variable, the operators $-, +, \times$ are respectively subtraction, addition, and product; p^n is exponentiation by a constant n, P is an arbitrary proposition. For $\mathbb{Q}$, equality is not Leibniz equality `=` but the equality of rationals `==`.

[314] Sources and binaries can be found at https://projects.coin-or.org/Csdp

For $\mathbb{Z}$ (resp. $\mathbb{Q}$), c ranges over integer constants (resp. rational constants). For $\mathbb{R}$, the tactic recognizes as real constants the following expressions:

```
c ::= R0 | R1 | Rmul(c,c) | Rplus(c,c) | Rminus(c,c) | IZR z | IQR q | Rdiv(c,c) | Rinv c
```

where z is a constant in $\mathbb{Z}$ and q is a constant in $\mathbb{Q}$. This includes integer constants written using the decimal notation, *i.e.*, `c%R`.

8.6.2 *Positivstellensatz* refutations

The name `psatz` is an abbreviation for *positivstellensatz* – literally ”positivity theorem” – which generalizes Hilbert’s *nullstellensatz*. It relies on the notion of Cone. Given a (finite) set of polynomials S, $Cone(S)$ is inductively defined as the smallest set of polynomials closed under the following rules:

$$\frac{p \in S}{p \in Cone(S)} \qquad \frac{}{p^2 \in Cone(S)} \qquad \frac{p_1 \in Cone(S) \quad p_2 \in Cone(S) \qquad \in \{+,*\}}{p_1 \quad p_2 \in Cone(S)}$$

The following theorem provides a proof principle for checking that a set of polynomial inequalities does not have solutions[315].

Theorem (Psatz). Let S be a set of polynomials. If -1 belongs to $Cone(S)$, then the conjunction $\bigwedge_{p\in S} p \geq 0$ is unsatisfiable. A proof based on this theorem is called a *positivstellensatz* refutation. The tactics work as follows. Formulas are normalized into conjunctive normal form $\bigwedge_i C_i$ where C_i has the general form $(\bigwedge_{j\in S_i} p_j \quad 0) \to False$ and $\quad \in \{>,\geq,=\}$ for $D \in \{\mathbb{Q},\mathbb{R}\}$ and $\quad \in \{\geq,=\}$ for $\mathbb{Z}$.

For each conjunct C_i, the tactic calls an oracle which searches for -1 within the cone. Upon success, the oracle returns a *cone expression* that is normalized by the *ring* tactic (see *The ring and field tactic families*) and checked to be -1.

8.6.3 `lra`: a decision procedure for linear real and rational arithmetic

`lra`

This tactic is searching for *linear* refutations. As a result, this tactic explores a subset of the *Cone* defined as

$$LinCone(S) = \left\{ \sum_{p\in S} \alpha_p \times p \;\middle|\; \alpha_p \text{ are positive constants} \right\}$$

The deductive power of *lra* overlaps with the one of *field* tactic *e.g.*, $x = 10 * x/10$ is solved by *lra*.

8.6.4 `lia`: a tactic for linear integer arithmetic

`lia`

This tactic solves linear goals over `Z` by searching for *linear* refutations and cutting planes. *lia* provides support for `Z`, `nat`, `positive` and `N` by pre-processing via the *zify* tactic.

High level view of `lia`

Over $\mathbb{R}$, *positivstellensatz* refutations are a complete proof principle[316]. However, this is not the case over $\mathbb{Z}$. Actually, *positivstellensatz* refutations are not even sufficient to decide linear *integer* arithmetic. The canonical example is $2 * x = 1 -> $ `False` which is a theorem of $\mathbb{Z}$ but not a theorem of $\mathbb{R}$. To remedy this weakness, the *lia* tactic is using recursively a combination of:

[315] Variants deal with equalities and strict inequalities.

[316] In practice, the oracle might fail to produce such a refutation.

- linear *positivstellensatz* refutations;
- cutting plane proofs;
- case split.

Cutting plane proofs

are a way to take into account the discreteness of $\mathbb{Z}$ by rounding up (rational) constants up-to the closest integer.

Theorem: `Bound on the ceiling function`

Let p be an integer and c a rational constant. Then $p \geq c \rightarrow p \geq \lceil c \rceil$.

For instance, from 2 x = 1 we can deduce

- $x \geq 1/2$ whose cut plane is $x \geq \lceil 1/2 \rceil = 1$;
- $x \leq 1/2$ whose cut plane is $x \leq \lfloor 1/2 \rfloor = 0$.

By combining these two facts (in normal form) $x - 1 \geq 0$ and $-x \geq 0$, we conclude by exhibiting a *positivstellensatz* refutation: $-1 \equiv x - 1 + -x \in Cone(x - 1, x)$.

Cutting plane proofs and linear *positivstellensatz* refutations are a complete proof principle for integer linear arithmetic.

Case split

enumerates over the possible values of an expression.

Theorem. Let p be an integer and c_1 and c_2 integer constants. Then:

$$c_1 \leq p \leq c_2 \Rightarrow \bigvee_{x \in [c_1, c_2]} p = x$$

Our current oracle tries to find an expression e with a small range $[c_1, c_2]$. We generate $c_2 - c_1$ subgoals which contexts are enriched with an equation $e = i$ for $i \in [c_1, c_2]$ and recursively search for a proof.

8.6.5 `nra`: a proof procedure for non-linear arithmetic

`nra`

This tactic is an *experimental* proof procedure for non-linear arithmetic. The tactic performs a limited amount of non-linear reasoning before running the linear prover of `lra`. This pre-processing does the following:

- If the context contains an arithmetic expression of the form $e[x^2]$ where x is a monomial, the context is enriched with $x^2 \geq 0$;
- For all pairs of hypotheses $e_1 \geq 0$, $e_2 \geq 0$, the context is enriched with $e_1 \times e_2 \geq 0$.

After this pre-processing, the linear prover of `lra` searches for a proof by abstracting monomials by variables.

8.6.6 `nia`: a proof procedure for non-linear integer arithmetic

`nia`

This tactic is a proof procedure for non-linear integer arithmetic. It performs a pre-processing similar to `nra`. The obtained goal is solved using the linear integer prover `lia`.

8.6.7 psatz: a proof procedure for non-linear arithmetic

psatz

This tactic explores the *Cone* by increasing degrees – hence the depth parameter n. In theory, such a proof search is complete – if the goal is provable the search eventually stops. Unfortunately, the external oracle is using numeric (approximate) optimization techniques that might miss a refutation.

To illustrate the working of the tactic, consider we wish to prove the following Coq goal:

```
Require Import ZArith Psatz.
Open Scope Z_scope.
Goal forall x, -x^2 >= 0 -> x - 1 >= 0 -> False.
intro x.
psatz Z 2.
```

As shown, such a goal is solved by `intro x. psatz Z 2.`. The oracle returns the cone expression $2 \times (x - 1) + (\mathbf{x} - \mathbf{1}) \times (\mathbf{x} - \mathbf{1}) + -x^2$ (polynomial hypotheses are printed in bold). By construction, this expression belongs to $\mathit{Cone}(-x^2, x - 1)$. Moreover, by running *ring* we obtain -1. By Theorem *Psatz*, the goal is valid.

8.6.8 zify: pre-processing of arithmetic goals

zify

This tactic is internally called by *lia* to support additional types e.g., `nat`, `positive` and `N`. By requiring the module `ZifyBool`, the boolean type `bool` and some comparison operators are also supported. *zify* can also be extended by rebinding the tactic `Zify.zify_post_hook` that is run immediately after *zify*.

- To support `Z.div` and `Z.modulo`: `Ltac Zify.zify_post_hook ::= Z.div_mod_to_equations`.
- To support `Z.quot` and `Z.rem`: `Ltac Zify.zify_post_hook ::= Z.quot_rem_to_equations`.
- To support `Z.div`, `Z.modulo`, `Z.quot`, and `Z.rem`: `Ltac Zify.zify_post_hook ::= Z.to_euclidean_division_equations`.

Command: `Show Zify InjTyp`

This command shows the list of types that can be injected into `Z`.

Command: `Show Zify BinOp`

This command shows the list of binary operators processed by *zify*.

Command: `Show Zify BinRel`

This command shows the list of binary relations processed by *zify*.

Command: `Show Zify UnOp`

This command shows the list of unary operators processed by *zify*.

Command: `Show Zify CstOp`

This command shows the list of constants processed by *zify*.

Command: `Show Zify Spec`

This command shows the list of operators over `Z` that are compiled using their specification e.g., `Z.min`.

8.7 Extraction of programs in OCaml and Haskell

Authors Jean-Christophe Filliâtre and Pierre Letouzey

We present here the Coq extraction commands, used to build certified and relatively efficient functional programs, extracting them from either Coq functions or Coq proofs of specifications. The functional languages available as output are currently OCaml, Haskell and Scheme. In the following, ”ML” will be used (abusively) to refer to any of the three.

Before using any of the commands or options described in this chapter, the extraction framework should first be loaded explicitly via `Require Extraction`, or via the more robust `From Coq Require Extraction`. Note that in earlier versions of Coq, these commands and options were directly available without any preliminary `Require`.

```
Require Extraction.
```

8.7.1 Generating ML Code

Note: In the following, a qualified identifier *qualid* can be used to refer to any kind of Coq global ”object” : constant, inductive type, inductive constructor or module name.

The next two commands are meant to be used for rapid preview of extraction. They both display extracted term(s) inside Coq.

Command: `Extraction` *qualid*

Extraction of the mentioned object in the Coq toplevel.

Command: `Recursive Extraction` *qualid*+

Recursive extraction of all the mentioned objects 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.

Command: `Extraction` *string* *qualid*+

Recursive extraction of all the mentioned objects and all their dependencies in one monolithic file *string*. Global and local identifiers are renamed according to the chosen ML language to fulfill its syntactic conventions, keeping original names as much as possible.

Command: `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.

Command: `Recursive Extraction Library` *ident*

Extraction of the Coq library *ident*`.v` and all other modules *ident*`.v` depends on.

Command: `Separate Extraction` *qualid*+

Recursive extraction of all the mentioned objects and all their dependencies, just as `Extraction` *string* *qualid*+, but instead of producing one monolithic file, this command splits the produced code in separate ML files, one per corresponding Coq `.v` file. This command is hence quite similar to *Recursive Extraction Library*, except that only the needed parts of Coq libraries are extracted instead of the whole. The naming convention in case of name clash is the same one as *Extraction Library*: identifiers are here renamed using prefixes `coq_` or `Coq_`.

The following command is meant to help automatic testing of the extraction, see for instance the `test-suite` directory in the Coq sources.

Command: `Extraction TestCompile` *qualid*+

All the mentioned objects and all their dependencies are extracted to a temporary OCaml file, just as in `Extraction "file"`. Then this temporary file and its signature are compiled with the same OCaml compiler used to built Coq. This command succeeds only if the extraction and the OCaml compilation succeed. It fails if the current target language of the extraction is not OCaml.

8.7.2 Extraction Options

Setting the target language

Command: `Extraction Language` `OCaml` | `Haskell` | `Scheme`

The ability to fix target language is the first and more important of the extraction options. Default is `OCaml`.

Inlining and optimizations

Since OCaml 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 wants to generate an OCaml program. The optimizations can be split in two groups: the type-preserving ones (essentially constant inlining and reductions) and the non type-preserving ones (some function abstractions of dummy types are removed when it is deemed safe in order to have more elegant types). Therefore some constants may not appear in the resulting monolithic OCaml program. In the case of modular extraction, even if some inlining is done, the inlined constants are nevertheless printed, to ensure session-independent programs.

Concerning Haskell, type-preserving optimizations are less useful because of laziness. We still make some optimizations, for example in order to produce more readable code.

The type-preserving optimizations are controlled by the following Coq flags and commands:

Flag: `Extraction Optimize`

Default is on. This controls all type-preserving optimizations made on the ML terms (mostly reduction of dummy beta/iota redexes, but also simplifications on Cases, etc). Turn this flag off if you want a ML term as close as possible to the Coq term.

Flag: `Extraction Conservative Types`

Default is off. This controls the non type-preserving optimizations made on ML terms (which try to avoid function abstraction of dummy types). Turn this flag on to make sure that `e:t` implies that `e':t'` where `e'` and `t'` are the extracted code of `e` and `t` respectively.

Flag: `Extraction KeepSingleton`

Default is off. Normally, when the extraction of an inductive type produces a singleton type (i.e. a type with only one constructor, and only one argument to this constructor), the inductive structure is removed and this type is seen as an alias to the inner type. The typical example is `sig`. This flag allows disabling this optimization when one wishes to preserve the inductive structure of types.

Flag: `Extraction AutoInline`

Default is on. The extraction mechanism inlines the bodies of some defined constants, according to some heuristics like size of bodies, uselessness of some arguments, etc. Those heuristics are not always perfect; if you want to disable this feature, turn this flag off.

Command: `Extraction Inline` *`qualid`*+

In addition to the automatic inline feature, the constants mentioned by this command will always be inlined during extraction.

Command: `Extraction NoInline` *qualid*+

Conversely, the constants mentioned by this command will never be inlined during extraction.

Command: `Print Extraction Inline`

Prints the current state of the table recording the custom inlinings declared by the two previous commands.

Command: `Reset Extraction Inline`

Empties the table recording the custom inlinings (see the previous commands).

Inlining and printing of a constant declaration:

The user can explicitly ask for 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-explicitly required but needed for dependency reasons, there are two cases:

- If an inlining decision is taken, whether automatically or not, all occurrences 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.

Extra elimination of useless arguments

The following command provides some extra manual control on the code elimination performed during extraction, in a way which is independent but complementary to the main elimination principles of extraction (logical parts and types).

Command: `Extraction Implicit` *qualid* `[` *ident*+ `]`

This experimental command allows declaring some arguments of *qualid* as implicit, i.e. useless in extracted code and hence to be removed by extraction. Here *qualid* can be any function or inductive constructor, and the given *ident* are the names of the concerned arguments. In fact, an argument can also be referred by a number indicating its position, starting from 1.

When an actual extraction takes place, an error is normally raised if the *Extraction Implicit* declarations cannot be honored, that is if any of the implicit arguments still occurs in the final code. This behavior can be relaxed via the following flag:

Flag: `Extraction SafeImplicits`

Default is on. When this flag is off, a warning is emitted instead of an error if some implicit arguments still occur in the final code of an extraction. This way, the extracted code may be obtained nonetheless and reviewed manually to locate the source of the issue (in the code, some comments mark the location of these remaining implicit arguments). Note that this extracted code might not compile or run properly, depending of the use of these remaining implicit arguments.

Realizing axioms

Extraction will fail if it encounters an informative axiom not realized. A warning will be issued if it encounters a logical axiom, to remind the user that inconsistent 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.

Command: `Extract Constant` *`qualid`* `=>` *`string`*

Give an ML extraction for the given constant. The *`string`* may be an identifier or a quoted string.

Command: `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:** This 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.

> **Caution:** 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 to the extracted files. The extraction recognizes whether the realized axiom should become a ML type constant or a ML object declaration. For example:

```
Axiom X:Set.
Axiom x:X.
Extract Constant X => "int".
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 have to be given (as quoted strings). The syntax is then:

Variant: `Extract Constant` *`qualid string`* `...` *`string`* `=>` *`string`*

The number of type variables is checked by the system. For example:

```
Axiom Y : Set -> Set -> Set.
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 has no computational content and hence will not appear 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 labeled `AXIOM TO BE REALIZED`. The user must then search these exceptions inside the extracted file and replace them by real code.

Realizing inductive types

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 the Coq one. The syntax is the following:

Command: `Extract Inductive` *`qualid`* `=>` *`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 (all the *`string`* between square brackets). In this form, the ML extraction must be an ML inductive datatype, and the native pattern matching of the language will be used.

Variant: `Extract Inductive` *qualid* `=>` *string* `[` *string*$^+$ `]` *string*

Same as before, with a final extra *string* that indicates how to perform pattern matching over this inductive type. In this form, the ML extraction could be an arbitrary type. For an inductive type with k constructors, the function used to emulate the pattern matching should expect $k+1$ arguments, first the k branches in functional form, and then the inductive element to destruct. For instance, the match branch `| S n => foo` gives the functional form `(fun n -> foo)`. Note that a constructor with no arguments is considered to have one unit argument, in order to block early evaluation of the branch: `| O => bar` leads to the functional form `(fun () -> bar)`. For instance, when extracting `nat` into OCaml `int`, the code to be provided has type: `(unit->'a)->(int->'a)->int->'a`.

Caution: As for *Extract Constant*, this command should be used with care:

- The ML code provided by the user is currently **not** checked at all by extraction, even for syntax errors.
- Extracting an inductive type to a pre-existing ML inductive type is quite sound. But extracting to a general type (by providing an ad-hoc pattern matching) will often **not** be fully rigorously correct. For instance, when extracting `nat` to OCaml `int`, it is theoretically possible to build `nat` values that are larger than OCaml `max_int`. It is the user's responsibility to be sure that no overflow or other bad events occur in practice.
- Translating an inductive type to an arbitrary ML type does **not** magically improve the asymptotic complexity of functions, even if the ML type is an efficient representation. For instance, when extracting `nat` to OCaml `int`, the function `Nat.mul` stays quadratic. It might be interesting to associate this translation with some specific *Extract Constant* when primitive counterparts exist.

Typical examples are the following:

```
Extract Inductive unit => "unit" [ "()" ].
Extract Inductive bool => "bool" [ "true" "false" ].
Extract Inductive sumbool => "bool" [ "true" "false" ].
```

Note: When extracting to OCaml, if an inductive constructor or type has arity 2 and the corresponding string is enclosed by parentheses, and the string meets OCaml's lexical criteria for an infix symbol, then the rest of the string is used as an infix constructor or type.

```
Extract Inductive list => "list" [ "[]" "(::)" ].
Extract Inductive prod => "(*)"  [ "(,)" ].
```

As an example of translation to a non-inductive datatype, let's turn `nat` into OCaml `int` (see caveat above):

```
Extract Inductive nat => int [ "0" "succ" ] "(fun fO fS n -> if n=0 then fO () else fS (n-1))".
```

Avoiding conflicts with existing filenames

When using *Extraction Library*, the names of the extracted files directly depend on the names of the Coq files. It may happen that these filenames are in conflict with already existing files, either in the standard library of the target language or in other code that is meant to be linked with the extracted code. For instance the module `List` exists both in Coq and in OCaml. It is possible to instruct the extraction not to use particular filenames.

Command: `Extraction Blacklist` *ident*+

Instruct the extraction to avoid using these names as filenames for extracted code.

Command: `Print Extraction Blacklist`

Show the current list of filenames the extraction should avoid.

Command: `Reset Extraction Blacklist`

Allow the extraction to use any filename.

For OCaml, a typical use of these commands is `Extraction Blacklist String List`.

Additional settings

Option: `Extraction File Comment` *string*

Provides a comment that is included at the beginning of the output files.

Option: `Extraction Flag` *num*

Controls which optimizations are used during extraction, providing a finer-grained control than *Extraction Optimize*. The bits of *num* are used as a bit mask. Keeping an option off keeps the extracted ML more similar to the Coq term. Values are:

Bit	Value	Optimization (default is on unless noted otherwise)
0	1	Remove local dummy variables
1	2	Use special treatment for fixpoints
2	4	Simplify case with iota-redux
3	8	Factor case branches as functions
4	16	(not available, default false)
5	32	Simplify case as function of one argument
6	64	Simplify case by swapping case and lambda
7	128	Some case optimization
8	256	Push arguments inside a letin
9	512	Use linear let reduction (default false)
10	1024	Use linear beta reduction (default false)

Flag: `Extraction TypeExpand`

If set, fully expand Coq types in ML. See the Coq source code to learn more.

8.7.3 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.

First, 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 alright but the generated code may be refused by the ML type checker. A very well known example is the `distr-pair` function:

```
Definition dp {A B:Type}(x:A)(y:B)(f:forall C:Type, 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))
```

Secondly, some Coq definitions 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 : Type := 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 (no GADT are produced yet by the extraction).

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 inductive types will always have the correct number of arguments, etc. Of course, when launching manually some extracted function, you should apply it to arguments of the right shape (from the Coq point-of-view).

More details about the correctness of the extracted programs can be found in *[Let02]*.

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 the OCaml type checker without any `Obj.magic` (see examples below).

8.7.4 Some examples

We present here two examples of extraction, taken from the Coq Standard Library. We choose OCaml as the target language, but everything, with slight modifications, can also be done in the other languages supported by extraction. We then indicate where to find other examples and tests of extraction.

A detailed example: Euclidean division

The file `Euclid` contains the proof of Euclidean division. The natural numbers used here are unary, represented by the type `nat`, which is defined by two constructors `O` and `S`. This module contains a theorem `eucl_dev`, whose type is:

```
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, plus some logical assertions that disappear during extraction. We can now extract this program to OCaml:

```
Require Extraction.
Require Import Euclid Wf_nat.
Extraction Inline gt_wf_rec lt_wf_rec induction_ltof2.
Recursive Extraction eucl_dev.
    type nat =
    | O
    | S of nat

    type sumbool =
    | Left
```

(continues on next page)

(continued from previous page)

```
| Right

(** val sub : nat -> nat -> nat **)

let rec sub n m =
  match n with
  | O -> n
  | S k -> (match m with
            | O -> n
            | S l -> sub k l)

(** val le_lt_dec : nat -> nat -> sumbool **)

let rec le_lt_dec n m =
  match n with
  | O -> Left
  | S n0 -> (match m with
             | O -> Right
             | S m0 -> le_lt_dec n0 m0)

(** val le_gt_dec : nat -> nat -> sumbool **)

let le_gt_dec =
  le_lt_dec

type diveucl =
| Divex of nat * nat

(** val eucl_dev : nat -> nat -> diveucl **)

let rec eucl_dev n m =
  let s = le_gt_dec n m in
  (match s with
   | Left ->
     let d = let y = sub m n in eucl_dev n y in
     let Divex (q, r) = d in Divex ((S q), r)
   | Right -> Divex (O, m))
```

The inlining of gt_wf_rec and others 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:

```
Extraction "euclid" eucl_dev.
```

Let us play the resulting program (in an OCaml toplevel):

```
#use "euclid.ml";;
type nat = O | S of nat
type sumbool = Left | Right
val sub : nat -> nat -> nat = <fun>
val le_lt_dec : nat -> nat -> sumbool = <fun>
val le_gt_dec : nat -> nat -> sumbool = <fun>
type diveucl = Divex of nat * nat
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 OCaml integers:

```
# let rec nat_of_int = function 0 -> O | n -> S (nat_of_int (n-1));;
val nat_of_int : int -> nat = <fun>

# let rec int_of_nat = function O -> 0 | S p -> 1+(int_of_nat p);;
val int_of_nat : nat -> int = <fun>

# let div a b =
  let Divex (q,r) = eucl_dev (nat_of_int b) (nat_of_int a)
  in (int_of_nat q, int_of_nat r);;
val div : int -> int -> int * int = <fun>

# div 173 15;;
- : int * int = (11, 8)
```

Note that these `nat_of_int` and `int_of_nat` are now available via a mere `Require Import ExtrOcamlIntConv` and then adding these functions to the list of functions to extract. This file `ExtrOcamlIntConv.v` and some others in `plugins/extraction/` are meant to help building concrete program via extraction.

Extraction's horror museum

Some pathological examples of extraction are grouped in the file `test-suite/success/extraction.v` of the sources of Coq.

Users' Contributions

Several of the Coq Users' Contributions use extraction to produce certified programs. In particular the following ones have an automatic extraction test:

- `additions` : https://github.com/coq-contribs/additions
- `bdds` : https://github.com/coq-contribs/bdds
- `canon-bdds` : https://github.com/coq-contribs/canon-bdds
- `chinese` : https://github.com/coq-contribs/chinese
- `continuations` : https://github.com/coq-contribs/continuations
- `coq-in-coq` : https://github.com/coq-contribs/coq-in-coq
- `exceptions` : https://github.com/coq-contribs/exceptions
- `firing-squad` : https://github.com/coq-contribs/firing-squad
- `founify` : https://github.com/coq-contribs/founify
- `graphs` : https://github.com/coq-contribs/graphs
- `higman-cf` : https://github.com/coq-contribs/higman-cf
- `higman-nw` : https://github.com/coq-contribs/higman-nw
- `hardware` : https://github.com/coq-contribs/hardware
- `multiplier` : https://github.com/coq-contribs/multiplier
- `search-trees` : https://github.com/coq-contribs/search-trees
- `stalmarck` : https://github.com/coq-contribs/stalmarck

Note that `continuations` and `multiplier` are a bit particular. They are examples of developments where `Obj.magic` is needed. This is probably due to a heavy use of impredicativity. After compilation, those two examples run nonetheless, thanks to the correction of the extraction *[Let02]*.

8.8 Program

Author Matthieu Sozeau

We present here the **Program** tactic commands, used to build certified Coq programs, elaborating them from their algorithmic skeleton and a rich specification *[Soz07]*. It can be thought of as a dual of *Extraction*. The goal of **Program** is to program as in a regular functional programming language whilst using as rich a specification as desired and proving that the code meets the specification using the whole Coq proof apparatus. This is done using a technique originating from the "Predicate subtyping" mechanism of PVS *[ROS98]*, which generates type checking conditions while typing a term constrained to a particular type. Here we insert existential variables in the term, which must be filled with proofs to get a complete Coq term. **Program** replaces the **Program** tactic by Catherine Parent *[Par95]* which had a similar goal but is no longer maintained.

The languages available as input are currently restricted to Coq's term language, but may be extended to OCaml, Haskell and others in the future. We use the same syntax as Coq and permit to use implicit arguments and the existing coercion mechanism. Input terms and types are typed in an extended system (Russell) and interpreted into Coq terms. The interpretation process may produce some proof obligations which need to be resolved to create the final term.

8.8.1 Elaborating programs

The main difference from Coq is that an object in a type `T : Set` can be considered as an object of type `{x : T | P}` for any well-formed `P : Prop`. If we go from `T` to the subset of `T` verifying property `P`, we must prove that the object under consideration verifies it. Russell will generate an obligation for every such coercion. In the other direction, Russell will automatically insert a projection.

Another distinction is the treatment of pattern matching. Apart from the following differences, it is equivalent to the standard match operation (see *Extended pattern matching*).

- Generation of equalities. A match expression is always generalized by the corresponding equality. As an example, the expression:

  ```
  match x with
  | 0 => t
  | S n => u
  end.
  ```

  will be first rewritten to:

  ```
  (match x as y return (x = y -> _) with
  | 0 => fun H : x = 0 -> t
  | S n => fun H : x = S n -> u
  end) (eq_refl x).
  ```

  This permits to get the proper equalities in the context of proof obligations inside clauses, without which reasoning is very limited.

- Generation of disequalities. If a pattern intersects with a previous one, a disequality is added in the context of the second branch. See for example the definition of div2 below, where the second branch is typed in a context where `p, _ <> S (S p)`.

- Coercion. If the object being matched is coercible to an inductive type, the corresponding coercion will be automatically inserted. This also works with the previous mechanism.

There are flags to control the generation of equalities and coercions.

Flag: `Program Cases`

This controls the special treatment of pattern matching generating equalities and disequalities when using **Program** (it is on by default). All pattern-matches and let-patterns are handled using the standard algorithm of Coq (see *Extended pattern matching*) when this flag is deactivated.

Flag: `Program Generalized Coercion`

This controls the coercion of general inductive types when using **Program** (the flag is on by default). Coercion of subset types and pairs is still active in this case.

Flag: `Program Mode`

Enables the program mode, in which 1) typechecking allows subset coercions and 2) the elaboration of pattern matching of *`Fixpoint`* and *`Definition`* act respectively like *`Program Fixpoint`* and *`Program Definition`*, generating obligations if there are unresolved holes after typechecking.

Syntactic control over equalities

To give more control over the generation of equalities, the type checker will fall back directly to Coq's usual typing of dependent pattern matching if a `return` or `in` clause is specified. Likewise, the if construct is not treated specially by **Program** so boolean tests in the code are not automatically reflected in the obligations. One can use the `dec` combinator to get the correct hypotheses as in:

```
Require Import Program Arith.
```

```
Program Definition id (n : nat) : { x : nat | x = n } :=
  if dec (leb n 0) then 0
  else S (pred n).
    id has type-checked, generating 2 obligations
    Solving obligations automatically...
    2 obligations remaining
    Obligation 1 of id:
    (forall n : nat, (n <=? 0) = true -> (fun x : nat => x = n) 0).

    Obligation 2 of id:
    (forall n : nat,
     (n <=? 0) = false -> (fun x : nat => x = n) (S (Init.Nat.pred n))).
```

The `let` tupling construct `let (x1, ..., xn) := t in b` does not produce an equality, contrary to the let pattern construct `let '(x1,..., xn) := t in b`. Also, `term :>` explicitly asks the system to coerce term to its support type. It can be useful in notations, for example:

```
Notation " x `= y " := (@eq _ (x :>) (y :>)) (only parsing).
```

This notation denotes equality on subset types using equality on their support types, avoiding uses of proof-irrelevance that would come up when reasoning with equality on the subset types themselves.

The next two commands are similar to their standard counterparts *`Definition`* and *`Fixpoint`* in that they define constants. However, they may require the user to prove some goals to construct the final definitions.

Program Definition

Command: `Program Definition` *`ident`* `:=` *`term`*

This command types the value term in Russell and generates proof obligations. Once solved using the commands shown below, it binds the final Coq term to the name `ident` in the environment.

Error: *`ident`* `already exists.`

Variant: `Program Definition` *`ident`* `:` *`type`* `:=` *`term`*

It interprets the type `type`, potentially generating proof obligations to be resolved. Once done with them, we have a Coq type $\texttt{type}_0$. It then elaborates the preterm `term` into a Coq term $\texttt{term}_0$, checking that the type of $\texttt{term}_0$ is coercible to $\texttt{type}_0$, and registers `ident` as being of type $\texttt{type}_0$ once the set of obligations generated during the interpretation of $\texttt{term}_0$ and the aforementioned coercion derivation are solved.

Error: `In environment … the term:` *`term`* `does not have type` *`type`*`. Actually, it has type ...`

Variant: `Program Definition` *`ident binders`* `:` *`type`* `:=` *`term`*

This is equivalent to:

`Program Definition ident : forall binders, type := fun binders => term`.

See also:

Sections *Controlling the reduction strategies and the conversion algorithm*, *`unfold`*

Program Fixpoint

Command: `Program Fixpoint` *`ident binders`* `{`*`order`*`}?` `:` *`type`* `:=` *`term`*

The optional order annotation follows the grammar:

```
order  ::=  measure term [ term ] | wf term ident
```

- `measure f R` where `f` is a value of type `X` computed on any subset of the arguments and the optional term `R` is a relation on `X`. `X` defaults to `nat` and `R` to `lt`.
- `wf R x` which is equivalent to `measure x R`.

The structural fixpoint operator behaves just like the one of Coq (see *`Fixpoint`*), except it may also generate obligations. It works with mutually recursive definitions too.

```
Require Import Program Arith.

Program Fixpoint div2 (n : nat) : { x : nat | n = 2 * x \/ n = 2 * x + 1 } :=
  match n with
  | S (S p) => S (div2 p)
  | _ => O
  end.
    Solving obligations automatically...
    4 obligations remaining
```

Here we have one obligation for each branch (branches for `0` and `(S 0)` are automatically generated by the pattern matching compilation algorithm).

```
Obligation 1.
    1 subgoal

      p, x : nat
      o : p = x + (x + 0) \/ p = x + (x + 0) + 1
```

(continues on next page)

(continued from previous page)

```
    ============================
    S (S p) = S (x + S (x + 0)) \/ S (S p) = S (x + S (x + 0) + 1)
```

One can use a well-founded order or a measure as termination orders using the syntax:

```
Program Fixpoint div2 (n : nat) {measure n} : { x : nat | n = 2 * x \/ n = 2 * x + 1 } :=
  match n with
  | S (S p) => S (div2 p)
  | _ => O
  end.
```

> **Caution:** When defining structurally recursive functions, the generated obligations should have the prototype of the currently defined functional in their context. In this case, the obligations should be transparent (e.g. defined using `Defined`) so that the guardedness condition on recursive calls can be checked by the kernel's type- checker. There is an optimization in the generation of obligations which gets rid of the hypothesis corresponding to the functional when it is not necessary, so that the obligation can be declared opaque (e.g. using `Qed`). However, as soon as it appears in the context, the proof of the obligation is *required* to be declared transparent.
>
> No such problems arise when using measures or well-founded recursion.

Program Lemma

Command: `Program Lemma` *`ident`* `:` *`type`*

The Russell language can also be used to type statements of logical properties. It will generate obligations, try to solve them automatically and fail if some unsolved obligations remain. In this case, one can first define the lemma's statement using `Program Definition` and use it as the goal afterwards. Otherwise the proof will be started with the elaborated version as a goal. The `Program` prefix can similarly be used as a prefix for `Variable`, `Hypothesis`, `Axiom` etc.

8.8.2 Solving obligations

The following commands are available to manipulate obligations. The optional identifier is used when multiple functions have unsolved obligations (e.g. when defining mutually recursive blocks). The optional tactic is replaced by the default one if not specified.

Command: `Local | Global`? `Obligation Tactic :=` *`tactic`*

Sets the default obligation solving tactic applied to all obligations automatically, whether to solve them or when starting to prove one, e.g. using `Next`. `Local` makes the setting last only for the current module. Inside sections, local is the default.

Command: `Show Obligation Tactic`

Displays the current default tactic.

Command: `Obligations` `of` *`ident`*?

Displays all remaining obligations.

Command: `Obligation` *`num`* `of` *`ident`*?

Start the proof of obligation *num*.

Command: `Next Obligation` `of` *`ident`* ?

Start the proof of the next unsolved obligation.

Command: `Solve Obligations` `of` *`ident`* ? `with` *`tactic`* ?

Tries to solve each obligation of `ident` using the given `tactic` or the default one.

Command: `Solve All Obligations` `with` *`tactic`* ?

Tries to solve each obligation of every program using the given tactic or the default one (useful for mutually recursive definitions).

Command: `Admit Obligations` `of` *`ident`* ?

Admits all obligations (of `ident`).

> **Note:** Does not work with structurally recursive programs.

Command: `Preterm` `of` *`ident`* ?

Shows the term that will be fed to the kernel once the obligations are solved. Useful for debugging.

Flag: `Transparent Obligations`

Controls whether all obligations should be declared as transparent (the default), or if the system should infer which obligations can be declared opaque.

Flag: `Hide Obligations`

Controls whether obligations appearing in the term should be hidden as implicit arguments of the special constantProgram.Tactics.obligation.

Flag: `Shrink Obligations`

Deprecated since version 8.7.

This flag (on by default) controls whether obligations should have their context minimized to the set of variables used in the proof of the obligation, to avoid unnecessary dependencies.

The module `Coq.Program.Tactics` defines the default tactic for solving obligations called `program_simpl`. Importing `Coq.Program.Program` also adds some useful notations, as documented in the file itself.

8.8.3 Frequently Asked Questions

Error: `Ill-formed recursive definition.`

This error can happen when one tries to define a function by structural recursion on a subset object, which means the Coq function looks like:

```
Program Fixpoint f (x : A | P) := match x with A b => f b end.
```

Supposing `b : A`, the argument at the recursive call to `f` is not a direct subterm of `x` as `b` is wrapped inside an `exist` constructor to build an object of type `{x : A | P}`. Hence the definition is rejected by the guardedness condition checker. However one can use wellfounded recursion on subset objects like this:

```
Program Fixpoint f (x : A | P) { measure (size x) } :=
  match x with A b => f b end.
```

One will then just have to prove that the measure decreases at each recursive call. There are three drawbacks though:

1. A measure function has to be defined;

2. The reduction is a little more involved, although it works well using lazy evaluation;
3. Mutual recursion on the underlying inductive type isn't possible anymore, but nested mutual recursion is always possible.

8.9 The ring and field tactic families

Author Bruno Barras, Benjamin Grégoire, Assia Mahboubi, Laurent Théry[317]

This chapter presents the tactics dedicated to dealing with ring and field equations.

8.9.1 What does this tactic do?

`ring` does associative-commutative rewriting in ring and semiring 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, ... and constants by application of $\oplus$ and $\otimes$.

Let an ordered product be a product of variables $V_{i_1} \otimes \cdots \otimes V_{i_n}$ verifying $i_1 \le i_2 \le \cdots \le i_n$. Let a monomial be the product of a constant and an ordered product. We can order the monomials by the lexicographic order on products of variables. Let a canonical sum be an ordered sum of monomials that are all different, i.e. each monomial in the sum is strictly less than the following monomial according to the lexicographic order. It is an easy theorem to show that every polynomial is equivalent (modulo the ring properties) to exactly one canonical sum. This canonical sum is called the normal form of the polynomial. In fact, the actual representation shares monomials with same prefixes. So what does the `ring` tactic do? It normalizes polynomials over any ring or semiring 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.

Example

In the ring of integers, the normal form of $x(3 + yx + 25(1 - z)) + zx$

is $28x + (-24)xz + xxy$.

`ring` is also able to compute a normal form modulo monomial equalities. For example, under the hypothesis that $2x^2 = yz + 1$, the normal form of $2(x + 1)x - x - zy$ is $x + 1$.

8.9.2 The variables map

It is frequent to have an expression built with + and ×, but rarely on variables only. Let us associate a number to each subterm of a ring expression in the Gallina language. For example, consider this expression in the semiring `nat`:

```
(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

[317] based on previous work from Patrick Loiseleur and Samuel Boutin

Then normalize the "abstract" polynomial $((V_1 \oplus V_2) \otimes V_2) \oplus (V_0 \otimes 2)$ 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)))
```

8.9.3 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.

8.9.4 Concrete usage in Coq

ring

This tactic solves equations upon polynomial expressions of a ring (or semiring) structure. It proceeds by normalizing both sides of the equation (w.r.t. associativity, commutativity and distributivity, constant propagation, rewriting of monomials) and comparing syntactically the results.

ring_simplify

This tactic applies the normalization procedure described above to the given terms. The tactic then replaces all occurrences of the terms given in the conclusion of the goal by their normal forms. If no term is given, then the conclusion should be an equation and both sides are normalized. The tactic can also be applied in a hypothesis.

The tactic must be loaded by `Require Import Ring`. The ring structures must be declared with the `Add Ring` command (see below). The ring of booleans is predefined; if one wants to use the tactic on `nat` one must first require the module `ArithRing` exported by `Arith`); for `Z`, do `Require Import ZArithRing` or simply `Require Import ZArith`; for `N`, do `Require Import NArithRing` or `Require Import NArith`.

Example

```
Require Import ZArith.
    [Loading ML file newring_plugin.cmxs ... done]
    [Loading ML file zify_plugin.cmxs ... done]
    [Loading ML file omega_plugin.cmxs ... done]

Open Scope Z_scope.
Goal forall a b c:Z,
    (a + b + c) ^ 2 =
     a * a + b ^ 2 + c * c + 2 * a * b + 2 * a * c + 2 * b * c.
    1 subgoal

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

intros; ring.
    No more subgoals.

Abort.
Goal forall a b:Z,
     2 * a * b = 30 -> (a + b) ^ 2 = a ^ 2 + b ^ 2 + 30.
```

(continues on next page)

(continued from previous page)

```
    1 subgoal

      ============================
      forall a b : Z, 2 * a * b = 30 -> (a + b) ^ 2 = a ^ 2 + b ^ 2 + 30

intros a b H; ring [H].
    No more subgoals.

Abort.
```

Variant: `ring [`*term**`]`

This tactic decides the equality of two terms modulo ring operations and the equalities defined by the *term*s. Each *term* has to be a proof of some equality `m = p`, where `m` is a monomial (after "abstraction"), `p` a polynomial and `=` the corresponding equality of the ring structure.

Variant: `ring_simplify [`*term**`]` *term** `in` *ident*

This tactic performs the simplification in the hypothesis named *ident*.

Note: `ring_simplify` *term*$_1$; `ring_simplify` *term*$_2$ is not equivalent to `ring_simplify` *term*$_1$ *term*$_2$.

In the latter case the variables map is shared between the two terms, and common subterm `t` of *term*$_1$ and *term*$_2$ will have the same associated variable number. So the first alternative should be avoided for terms belonging to the same ring theory.

Error messages:

Error: `Not a valid ring equation.`

The conclusion of the goal is not provable in the corresponding ring theory.

Error: `Arguments of ring_simplify do not have all the same type.`

ring_simplify cannot simplify terms of several rings at the same time. Invoke the tactic once per ring structure.

Error: `Cannot find a declared ring structure over` *term*.

No ring has been declared for the type of the terms to be simplified. Use *Add Ring* first.

Error: `Cannot find a declared ring structure for equality` *term*.

Same as above in the case of the *ring* tactic.

8.9.5 Adding a ring structure

Declaring a new ring consists in proving that a ring signature (a carrier set, an equality, and ring operations: `Ring_theory.ring_theory` and `Ring_theory.semi_ring_theory`) satisfies the ring axioms. Semi- rings (rings without + inverse) are also supported. The equality can be either Leibniz equality, or any relation declared as a setoid (see *Tactics enabled on user provided relations*). The definitions of ring and semiring (see module `Ring_theory`) are:

```
Record ring_theory : Prop := mk_rt {
  Radd_0_l    : forall x, 0 + x == x;
  Radd_sym    : forall x y, x + y == y + x;
  Radd_assoc  : forall x y z, x + (y + z) == (x + y) + z;
  Rmul_1_l    : forall x, 1 * x == x;
  Rmul_sym    : forall x y, x * y == y * x;
```

(continues on next page)

(continued from previous page)

```
  Rmul_assoc  : forall x y z, x * (y * z) == (x * y) * z;
  Rdistr_l    : forall x y z, (x + y) * z == (x * z) + (y * z);
  Rsub_def    : forall x y, x - y == x + -y;
  Ropp_def    : forall x, x + (- x) == 0
}.

Record semi_ring_theory : Prop := mk_srt {
  SRadd_0_l   : forall n, 0 + n == n;
  SRadd_sym   : forall n m, n + m == m + n ;
  SRadd_assoc : forall n m p, n + (m + p) == (n + m) + p;
  SRmul_1_l   : forall n, 1*n == n;
  SRmul_0_l   : forall n, 0*n == 0;
  SRmul_sym   : forall n m, n*m == m*n;
  SRmul_assoc : forall n m p, n*(m*p) == (n*m)*p;
  SRdistr_l   : forall n m p, (n + m)*p == n*p + m*p
}.
```

This implementation of `ring` also features a notion of constant that can be parameterized. This can be used to improve the handling of closed expressions when operations are effective. It consists in introducing a type of *coefficients* and an implementation of the ring operations, and a morphism from the coefficient type to the ring carrier type. The morphism needs not be injective, nor surjective.

As an example, one can consider the real numbers. The set of coefficients could be the rational numbers, upon which the ring operations can be implemented. The fact that there exists a morphism is defined by the following properties:

```
Record ring_morph : Prop := mkmorph {
  morph0    : [cO] == 0;
  morph1    : [cI] == 1;
  morph_add : forall x y, [x +! y] == [x]+[y];
  morph_sub : forall x y, [x -! y] == [x]-[y];
  morph_mul : forall x y, [x *! y] == [x]*[y];
  morph_opp : forall x, [-!x] == -[x];
  morph_eq  : forall x y, x?=!y = true -> [x] == [y]
}.

Record semi_morph : Prop := mkRmorph {
  Smorph0 : [cO] == 0;
  Smorph1 : [cI] == 1;
  Smorph_add : forall x y, [x +! y] == [x]+[y];
  Smorph_mul : forall x y, [x *! y] == [x]*[y];
  Smorph_eq  : forall x y, x?=!y = true -> [x] == [y]
}.
```

where `cO` and `cI` denote the 0 and 1 of the coefficient set, `+!`, `*!`, `-!` are the implementations of the ring operations, `==` is the equality of the coefficients, `?+!` is an implementation of this equality, and `[x]` is a notation for the image of `x` by the ring morphism.

Since `Z` is an initial ring (and `N` is an initial semiring), it can always be considered as a set of coefficients. There are basically three kinds of (semi-)rings:

abstract rings to be used when operations are not effective. The set of coefficients is `Z` (or `N` for semirings).

computational rings to be used when operations are effective. The set of coefficients is the ring itself. The user only has to provide an implementation for the equality.

customized ring for other cases. The user has to provide the coefficient set and the morphism.

This implementation of ring can also recognize simple power expressions as ring expressions. A power function is specified by the following property:

```
Require Import Reals.
Section POWER.
Variable Cpow : Set.
Variable Cp_phi : N -> Cpow.
Variable rpow : R -> Cpow -> R.
Record power_theory : Prop := mkpow_th {
    rpow_pow_N : forall r n, rpow r (Cp_phi n) = pow_N 1%R Rmult r n
  }.
End POWER.
```

The syntax for adding a new ring is

Command: `Add Ring` *ident* : *term* (*ring_mod* , *ring_mod* *) ?

The *ident* is not relevant. It is used just for error messages. The *term* is a proof that the ring signature satisfies the (semi-)ring axioms. The optional list of modifiers is used to tailor the behavior of the tactic. The following list describes their syntax and effects:

```
ring_mod  ::=  abstract | decidable term | morphism term
               setoid term term
               constants [ tactic ]
               preprocess [ tactic ]
               postprocess [ tactic ]
               power_tac term [ tactic ]
               sign term
               div term
```

abstract declares the ring as abstract. This is the default.

decidable *term* declares the ring as computational. The expression *term* is the correctness proof of an equality test `?=!` (which should be evaluable). Its type should be of the form `forall x y, x ?=! y = true → x == y`.

morphism *term* declares the ring as a customized one. The expression *term* is a proof that there exists a morphism between a set of coefficient and the ring carrier (see `Ring_theory.ring_morph` and `Ring_theory.semi_morph`).

setoid *term* *term* forces the use of given setoid. The first *term* is a proof that the equality is indeed a setoid (see `Setoid.Setoid_Theory`), and the second *term* a proof that the ring operations are morphisms (see `Ring_theory.ring_eq_ext` and `Ring_theory.sring_eq_ext`). This modifier needs not be used if the setoid and morphisms have been declared.

constants [*tactic*] specifies a tactic expression *tactic* that, given a term, returns either an object of the coefficient set that is mapped to the expression via the morphism, or returns `InitialRing.NotConstant`. The default behavior is to map only 0 and 1 to their counterpart in the coefficient set. This is generally not desirable for non trivial computational rings.

preprocess [*tactic*] specifies a tactic *tactic* that is applied as a preliminary step for *ring* and *ring_simplify*. It can be used to transform a goal so that it is better recognized. For instance, `S n` can be changed to `plus 1 n`.

postprocess [*tactic*] specifies a tactic *tactic* that is applied as a final step for *ring_simplify*. For instance, it can be used to undo modifications of the preprocessor.

power_tac *term* [*tactic*] allows *ring* and *ring_simplify* to recognize power expressions with a constant positive integer exponent (example: x^2). The term *term* is a proof that a given power function satisfies the specification of a power function (term has to be a proof of `Ring_theory.power_theory`) and *tactic* specifies a tactic expression that, given a term, “abstracts” it into an object of type `N` whose interpretation via `Cp_phi` (the evaluation function of power coefficient) is the original term, or returns `InitialRing.NotConstant` if not a constant coefficient (i.e. L$_{\text{tac}}$ is the inverse function of `Cp_phi`). See files `plugins/setoid_ring/ZArithRing.v` and `plugins/setoid_ring/RealField.v` for examples. By default the tactic does not recognize power expressions as ring expressions.

sign *term* allows *ring_simplify* to use a minus operation when outputting its normal form, i.e writing `x - y` instead of `x + (- y)`. The term *term* is a proof that a given sign function indicates expressions that are signed (*term* has to be a proof of `Ring_theory.get_sign`). See `plugins/setoid_ring/InitialRing.v` for examples of sign function.

div *term* allows *ring* and *ring_simplify* to use monomials with coefficients other than 1 in the rewriting. The term *term* is a proof that a given division function satisfies the specification of an euclidean division function (*term* has to be a proof of `Ring_theory.div_theory`). For example, this function is called when trying to rewrite $7x$ by $2x = z$ to tell that $7 = 3 \times 2 + 1$. See `plugins/setoid_ring/InitialRing.v` for examples of div function.

Error messages:

Error: **`Bad ring structure.`**

The proof of the ring structure provided is not of the expected type.

Error: **`Bad lemma for decidability of equality.`**

The equality function provided in the case of a computational ring has not the expected type.

Error: **`Ring operation should be declared as a morphism.`**

A setoid associated to the carrier of the ring structure has been found, but the ring operation should be declared as morphism. See *Tactics enabled on user provided relations*.

8.9.6 How does it work?

The code of `ring` is a good example of a tactic written using *reflection*. What is reflection? Basically, using it means that a part of a tactic is written in Gallina, Coq’s language of terms, rather than L$_{\text{tac}}$ or OCaml. From the philosophical point of view, reflection 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 its correctness.

The interested reader is strongly advised to have a look at the file `Ring_polynom.v`. Here a type for polynomials is defined:

```
Inductive PExpr : Type :=
  | PEc : C -> PExpr
  | PEX : positive -> PExpr
  | PEadd : PExpr -> PExpr -> PExpr
  | PEsub : PExpr -> PExpr -> PExpr
  | PEmul : PExpr -> PExpr -> PExpr
  | PEopp : PExpr -> PExpr
  | PEpow : PExpr -> N -> PExpr.
```

Polynomials in normal form are defined as:

```
Inductive Pol : Type :=
  | Pc : C -> Pol
  | Pinj : positive -> Pol -> Pol
  | PX : Pol -> positive -> Pol -> Pol.
```

where `Pinj n P` denotes `P` in which V_i is replaced by V_{i+n} , and `PX P n Q` denotes $P \otimes V_1^n \oplus Q'$, `Q'` being `Q` where V_i is replaced by V_{i+1}.

Variable maps are represented by lists of ring elements, and two interpretation functions, one that maps a variables map and a polynomial to an element of the concrete ring, and the second one that does the same for normal forms:

```
Definition PEeval : list R -> PExpr -> R := [...].
Definition Pphi_dev : list R -> Pol -> R := [...].
```

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

```
Definition norm : PExpr -> Pol := [...].
Lemma Pphi_dev_ok :
   forall l pe npe, norm pe = npe -> PEeval l pe == Pphi_dev l npe.
```

So now, what is the scheme for a normalization proof? Let p be the polynomial expression that the user wants to normalize. First a little piece of ML code guesses the type of `p`, the ring theory `T` to use, an abstract polynomial `ap` and a variables map `v` such that `p` is $\beta\delta\iota$- equivalent to `(PEeval v ap)`. Then we replace it by `(Pphi_dev v (norm ap))`, using the main correctness theorem and we reduce it to a concrete expression `p'`, which is the concrete normal form of `p`. This is summarized in this diagram:

p	$\rightarrow_{\beta\delta\iota}$	(PEeval v ap)
	$=_{\text{(by the main correctness theorem)}}$	
p'	$\leftarrow_{\beta\delta\iota}$	(Pphi_dev v (norm ap))

The user does not see the right part of the diagram. From outside, the tactic behaves like a $\beta\delta\iota$ simplification extended with rewriting rules for associativity and commutativity. Basically, the proof is only the application of the main correctness theorem to well-chosen arguments.

8.9.7 Dealing with fields

field

This tactic is an extension of the *ring* tactic that deals with rational expressions. Given a rational expression $F = 0$. It first reduces the expression `F` to a common denominator $N/D = 0$ where `N` and `D` are two ring expressions. For example, if we take $F = (1-1/x)x-x+1$, this gives $N = (x-1)x-x^2+x$ and $D = x$. It then calls ring to solve $N = 0$.

Note that `field` also generates nonzero conditions for all the denominators it encounters in the reduction. In our example, it generates the condition $x \neq 0$. These conditions appear as one subgoal which is a conjunction if there are several denominators. Nonzero conditions are always polynomial expressions. For example when reducing the expression $1/(1+1/x)$, two side conditions are generated: $x \neq 0$ and $x+1 \neq 0$. Factorized expressions are broken since a field is an integral domain, and when the equality test on coefficients is complete w.r.t. the equality of the target field, constants can be proven different from zero automatically.

The tactic must be loaded by `Require Import Field`. New field structures can be declared to the system with the `Add Field` command (see below). The field of real numbers is defined in module `RealField` (in `plugins/setoid_ring`). It is exported by module `Rbase`, so that requiring `Rbase` or

Reals is enough to use the field tactics on real numbers. Rational numbers in canonical form are also declared as a field in the module Qcanon.

Example

```
Require Import Reals.
Open Scope R_scope.
Goal forall x,
       x <> 0 -> (1 - 1 / x) * x - x + 1 = 0.
    1 subgoal

      ============================
      forall x : R, x <> 0 -> (1 - 1 / x) * x - x + 1 = 0

intros; field; auto.
    No more subgoals.

Abort.
Goal forall x y,
       y <> 0 -> y = x -> x / y = 1.
    1 subgoal

      ============================
      forall x y : R, y <> 0 -> y = x -> x / y = 1

intros x y H H1; field [H1]; auto.
    No more subgoals.

Abort.
```

Variant: `field [` *term*`*` `]`

This tactic decides the equality of two terms modulo field operations and the equalities defined by the *terms*. Each *term* has to be a proof of some equality `m = p`, where `m` is a monomial (after "abstraction"), `p` a polynomial and `=` the corresponding equality of the field structure.

Note: Rewriting works with the equality `m = p` only if `p` is a polynomial since rewriting is handled by the underlying ring tactic.

Variant: `field_simplify`

performs the simplification in the conclusion of the goal, $F_1 = F_2$ becomes $N_1/D_1 = N_2/D_2$. A normalization step (the same as the one for rings) is then applied to N_1, D_1, N_2 and D_2. This way, polynomials remain in factorized form during fraction simplification. This yields smaller expressions when reducing to the same denominator since common factors can be canceled.

Variant: `field_simplify [` *term*`*` `]`

This variant performs the simplification in the conclusion of the goal using the equalities defined by the *terms*.

Variant: `field_simplify [` *term*`*` `]` *term*`*`

This variant performs the simplification in the terms *terms* of the conclusion of the goal using the equalities defined by *terms* inside the brackets.

Variant: `field_simplify in` *ident*

This variant performs the simplification in the assumption *ident*.

Variant: `field_simplify [term*] in ident`

This variant performs the simplification in the assumption *ident* using the equalities defined by the *terms*.

Variant: `field_simplify [term*] term* in ident`

This variant performs the simplification in the *terms* of the assumption *ident* using the equalities defined by the *terms* inside the brackets.

Variant: `field_simplify_eq`

performs the simplification in the conclusion of the goal removing the denominator. $F_1 = F_2$ becomes $N_1 D_2 = N_2 D_1$.

Variant: `field_simplify_eq [ term*]`

This variant performs the simplification in the conclusion of the goal using the equalities defined by *terms*.

Variant: `field_simplify_eq in ident`

This variant performs the simplification in the assumption *ident*.

Variant: `field_simplify_eq [term*] in ident`

This variant performs the simplification in the assumption *ident* using the equalities defined by *terms* and removing the denominator.

8.9.8 Adding a new field structure

Declaring a new field consists in proving that a field signature (a carrier set, an equality, and field operations: `Field_theory.field_theory` and `Field_theory.semi_field_theory`) satisfies the field axioms. Semi-fields (fields without + inverse) are also supported. The equality can be either Leibniz equality, or any relation declared as a setoid (see *Tactics enabled on user provided relations*). The definition of fields and semifields is:

```
Record field_theory : Prop := mk_field {
  F_R : ring_theory rO rI radd rmul rsub ropp req;
  F_1_neq_0 : ~ 1 == 0;
  Fdiv_def : forall p q, p / q == p * / q;
  Finv_l : forall p, ~ p == 0 ->  / p * p == 1
}.

Record semi_field_theory : Prop := mk_sfield {
  SF_SR : semi_ring_theory rO rI radd rmul req;
  SF_1_neq_0 : ~ 1 == 0;
  SFdiv_def : forall p q, p / q == p * / q;
  SFinv_l : forall p, ~ p == 0 ->  / p * p == 1
}.
```

The result of the normalization process is a fraction represented by the following type:

```
Record linear : Type := mk_linear {
  num : PExpr C;
  denum : PExpr C;
  condition : list (PExpr C)
}.
```

where `num` and `denum` are the numerator and denominator; `condition` is a list of expressions that have appeared as a denominator during the normalization process. These expressions must be proven different from zero for the correctness of the algorithm.

The syntax for adding a new field is

Command: `Add Field` *ident* `:` *term* (`(` *field_mod* (`,` *field_mod*)* `)`)?

The *ident* is not relevant. It is used just for error messages. *term* is a proof that the field signature satisfies the (semi-)field axioms. The optional list of modifiers is used to tailor the behavior of the tactic.

field_mod ::= *ring_mod* | completeness *term*

Since field tactics are built upon `ring` tactics, all modifiers of the `Add Ring` apply. There is only one specific modifier:

completeness *term* allows the field tactic to prove automatically that the image of nonzero coefficients are mapped to nonzero elements of the field. *term* is a proof of `forall x y, [x] == [y] -> x ?=! y = true`, which is the completeness of equality on coefficients w.r.t. the field equality.

8.9.9 History of ring

First Samuel Boutin designed the tactic `ACDSimpl`. This tactic did lot of rewriting. But the proofs terms generated by rewriting were too big for Coq's type checker. Let us see why:

```
Require Import ZArith.
Open Scope Z_scope.
Goal forall x y z : Z,
       x + 3 + y + y * z = x + 3 + y + z * y.
    1 subgoal

      ============================
      forall x y z : Z, x + 3 + y + y * z = x + 3 + y + z * y

intros; rewrite (Zmult_comm y z); reflexivity.
    No more subgoals.

Save foo.
Print foo.
    foo =
    fun x y z : Z =>
    eq_ind_r (fun z0 : Z => x + 3 + y + z0 = x + 3 + y + z * y) eq_refl
      (Z.mul_comm y z)
         : forall x y z : Z, x + 3 + y + y * z = x + 3 + y + z * y

    Arguments foo (_ _ _)%Z_scope
```

At each step of rewriting, the whole context is duplicated in the proof term. Then, a tactic that does hundreds of rewriting generates huge proof terms. Since `ACDSimpl` was too slow, Samuel Boutin rewrote it using reflection (see [Bou97]). Later, it was rewritten by Patrick Loiseleur: the new tactic does not any more require `ACDSimpl` to compile and it makes use of $\beta\delta\iota$-reduction not only to replace the rewriting steps, but also to achieve the interleaving of computation and reasoning (see *Discussion*). He also wrote some ML code for the `Add Ring` command that allows registering new rings dynamically.

Proofs terms generated by ring are quite small, they are linear in the number of $\oplus$ and $\otimes$ operations in the normalized terms. Type checking those terms requires some time because it makes a large use of the conversion rule, but memory requirements are much smaller.

8.9.10 Discussion

Efficiency is not the only motivation to use reflection here. `ring` also deals with constants, it rewrites for example the expression `34 + 2 * x - x + 12` to the expected result `x + 46`. For the tactic `ACDSimpl`, the only constants were 0 and 1. So the expression `34 + 2 * (x - 1) + 12` is interpreted as $V_0 \oplus V_1 \otimes (V_2 \ominus 1) \oplus V_3$, with the variables mapping $\{V_0 \mapsto 34; V_1 \mapsto 2; V_2 \mapsto x; V_3 \mapsto 12\}$. Then it is rewritten to `34 - x + 2 * x + 12`, very far from the expected result. Here rewriting is not sufficient: you have to do some kind of reduction (some kind of computation) to achieve the normalization.

The tactic `ring` is not only faster than the old one: by using reflection, we get for free the integration of computation and reasoning that would be very difficult to implement without it.

Is it the ultimate way to write tactics? The answer is: yes and no. The `ring` tactic intensively uses the conversion rules of the Calculus of Inductive Constructions, i.e. it replaces proofs by computations as much as possible. It can be useful in all situations where a classical tactic generates huge proof terms, like symbolic processing and tautologies. But there are also tactics like `auto` or `linear` that do many complex computations, using side-effects and backtracking, and generate a small proof term. Clearly, it would be significantly less efficient to replace them by tactics using reflection.

Another idea suggested by Benjamin Werner: reflection could be used to couple an external tool (a rewriting program or a model checker) with Coq. We define (in Coq) a type of terms, a type of *traces*, and prove a correctness theorem that states that *replaying traces* is safe with respect to some interpretation. Then we let the external tool do every computation (using side-effects, backtracking, exception, or others features that are not available in pure lambda calculus) to produce the trace. Now we can check in Coq that the trace has the expected semantics by applying the correctness theorem.

8.10 Nsatz: tactics for proving equalities in integral domains

Author Loïc Pottier

nsatz

This tactic is for solving goals of the form

$$\begin{array}{l}\forall X_1, \ldots, X_n \in A, \\ P_1(X_1, \ldots, X_n) = Q_1(X_1, \ldots, X_n), \ldots, P_s(X_1, \ldots, X_n) = Q_s(X_1, \ldots, X_n) \\ \vdash P(X_1, \ldots, X_n) = Q(X_1, \ldots, X_n)\end{array}$$

where $P, Q, P_1, Q_1, \ldots, P_s, Q_s$ are polynomials and A is an integral domain, i.e. a commutative ring with no zero divisors. For example, A can be $\mathbb{R}$, $\mathbb{Z}$, or $\mathbb{Q}$. Note that the equality $=$ used in these goals can be any setoid equality (see *Tactics enabled on user provided relations*) , not only Leibniz equality.

It also proves formulas

$$\begin{array}{l}\forall X_1, \ldots, X_n \in A, \\ P_1(X_1, \ldots, X_n) = Q_1(X_1, \ldots, X_n) \wedge \ldots \wedge P_s(X_1, \ldots, X_n) = Q_s(X_1, \ldots, X_n) \\ \rightarrow P(X_1, \ldots, X_n) = Q(X_1, \ldots, X_n)\end{array}$$

doing automatic introductions.

You can load the `Nsatz` module with the command `Require Import Nsatz`.

8.10.1 More about `nsatz`

Hilbert's Nullstellensatz theorem shows how to reduce proofs of equalities on polynomials on a commutative ring A with no zero divisors to algebraic computations: it is easy to see that if a polynomial P in $A[X_1, \ldots, X_n]$ verifies $cP^r = \sum_{i=1}^{s} S_i P_i$, with $c \in A$, $c \neq 0$, r a positive integer, and the S_i s in $A[X_1, \ldots, X_n]$, then P is zero whenever polynomials $P_1, \dots, P_s$ are zero (the converse is also true when A is an algebraically closed field: the method is complete).

So, solving our initial problem reduces to finding $S_1, \dots, S_s$, c and r such that $c(P-Q)^r = \sum_i S_i(P_i - Q_i)$, which will be proved by the tactic ring.

This is achieved by the computation of a Gröbner basis of the ideal generated by $P_1 - Q_1, \dots, P_s - Q_s$, with an adapted version of the Buchberger algorithm.

This computation is done after a step of *reification*, which is performed using *Typeclasses*.

Variant: `nsatz with radicalmax:=num%N strategy:=num%Z parameters:=[ident*,] variables:=[ident*,]`

Most complete syntax for `nsatz`.

- `radicalmax` is a bound when searching for r such that $c(P-Q)r = \sum_{i=1..s} S_i(Pi - Qi)$
- `strategy` gives the order on variables $X_1, \dots, X_n$ and the strategy used in Buchberger algorithm (see *[GMN+91]* for details):
  - strategy = 0: reverse lexicographic order and newest s-polynomial.
  - strategy = 1: reverse lexicographic order and sugar strategy.
  - strategy = 2: pure lexicographic order and newest s-polynomial.
  - strategy = 3: pure lexicographic order and sugar strategy.
- `parameters` is the list of variables $X_{i_1}, \dots, X_{i_k}$ among $X_1, \dots, X_n$ which are considered as parameters: computation will be performed with rational fractions in these variables, i.e. polynomials are considered with coefficients in $R(X_{i_1}, \dots, X_{i_k})$. In this case, the coefficient c can be a non constant polynomial in $X_{i_1}, \dots, X_{i_k}$, and the tactic produces a goal which states that c is not zero.
- `variables` is the list of the variables in the decreasing order in which they will be used in the Buchberger algorithm. If `variables` = `(@nil R)`, then `lvar` is replaced by all the variables which are not in `parameters`.

See the file `Nsatz.v` for many examples, especially in geometry.

8.11 Generalized rewriting

Author Matthieu Sozeau

This chapter presents the extension of several equality related tactics to work over user-defined structures (called setoids) that are equipped with ad-hoc equivalence relations meant to behave as equalities. Actually, the tactics have also been generalized to relations weaker than equivalences (e.g. rewriting systems). The toolbox also extends the automatic rewriting capabilities of the system, allowing the specification of custom strategies for rewriting.

This documentation is adapted from the previous setoid documentation by Claudio Sacerdoti Coen (based on previous work by Clément Renard). The new implementation is a drop-in replacement for the old one[318], hence most of the documentation still applies.

The work is a complete rewrite of the previous implementation, based on the typeclass infrastructure. It also improves on and generalizes the previous implementation in several ways:

- User-extensible algorithm. The algorithm is separated into two parts: generation of the rewriting constraints (written in ML) and solving these constraints using typeclass resolution. As typeclass resolution is extensible using tactics, this allows users to define general ways to solve morphism constraints.

[318] Nicolas Tabareau helped with the gluing.

- Subrelations. An example extension to the base algorithm is the ability to define one relation as a subrelation of another so that morphism declarations on one relation can be used automatically for the other. This is done purely using tactics and typeclass search.
- Rewriting under binders. It is possible to rewrite under binders in the new implementation, if one provides the proper morphisms. Again, most of the work is handled in the tactics.
- First-class morphisms and signatures. Signatures and morphisms are ordinary Coq terms, hence they can be manipulated inside Coq, put inside structures and lemmas about them can be proved inside the system. Higher-order morphisms are also allowed.
- Performance. The implementation is based on a depth-first search for the first solution to a set of constraints which can be as fast as linear in the size of the term, and the size of the proof term is linear in the size of the original term. Besides, the extensibility allows the user to customize the proof search if necessary.

8.11.1 Introduction to generalized rewriting

Relations and morphisms

A parametric *relation* `R` is any term of type `forall (x1 : T1) ... (xn : Tn), relation A`. The expression `A`, which depends on `x1 ... xn` , is called the *carrier* of the relation and `R` is said to be a relation over `A`; the list `x1,...,xn` is the (possibly empty) list of parameters of the relation.

Example: Parametric relation

It is possible to implement finite sets of elements of type `A` as unordered lists of elements of type `A`. The function `set_eq: forall (A : Type), relation (list A)` satisfied by two lists with the same elements is a parametric relation over `(list A)` with one parameter `A`. The type of `set_eq` is convertible with `forall (A : Type), list A -> list A -> Prop`.

An *instance* of a parametric relation `R` with n parameters is any term `(R t1 ... tn)`.

Let `R` be a relation over `A` with `n` parameters. A term is a parametric proof of reflexivity for `R` if it has type `forall (x1 : T1) ... (xn : Tn), reflexive (R x1 ... xn)`. Similar definitions are given for parametric proofs of symmetry and transitivity.

Example: Parametric relation (continued)

The `set_eq` relation of the previous example can be proved to be reflexive, symmetric and transitive. A parametric unary function `f` of type `forall (x1 : T1) ... (xn : Tn), A1 -> A2` covariantly respects two parametric relation instances `R1` and `R2` if, whenever `x, y` satisfy `R1 x y`, their images (`f x`) and (`f y`) satisfy `R2 (f x) (f y)`. An `f` that respects its input and output relations will be called a unary covariant *morphism*. We can also say that `f` is a monotone function with respect to `R1` and `R2` . The sequence `x1 ... xn` represents the parameters of the morphism.

Let `R1` and `R2` be two parametric relations. The *signature* of a parametric morphism of type `forall (x1 : T1) ... (xn : Tn), A1 -> A2` that covariantly respects two instances I_{R_1} and I_{R_2} of `R1` and `R2` is written $I_{R_1} {++}{>} I_{R_2}$. Notice that the special arrow ++>, which reminds the reader of covariance, is placed between the two relation instances, not between the two carriers. The signature relation instances and morphism will be typed in a context introducing variables for the parameters.

The previous definitions are extended straightforwardly to n-ary morphisms, that are required to be simultaneously monotone on every argument.

Morphisms can also be contravariant in one or more of their arguments. A morphism is contravariant on an argument associated to the relation instance R if it is covariant on the same argument when the inverse relation R^{-1} (`inverse R` in Coq) is considered. The special arrow `-->` is used in signatures for contravariant morphisms.

Functions having arguments related by symmetric relations instances are both covariant and contravariant in those arguments. The special arrow `==>` is used in signatures for morphisms that are both covariant and contravariant.

An instance of a parametric morphism f with n parameters is any term $f\, t_1 \ldots t_n$.

Example: Morphisms

Continuing the previous example, let `union: forall (A : Type), list A -> list A -> list A` perform the union of two sets by appending one list to the other. `union` is a binary morphism parametric over `A` that respects the relation instance `(set_eq A)`. The latter condition is proved by showing:

```
forall (A: Type) (S1 S1' S2 S2': list A),
  set_eq A S1 S1' ->
  set_eq A S2 S2' ->
  set_eq A (union A S1 S2) (union A S1' S2').
```

The signature of the function `union A` is `set_eq A ==> set_eq A ==> set_eq A` for all `A`.

Example: Contravariant morphisms

The division function `Rdiv : R -> R -> R` is a morphism of signature `le ++> le --> le` where `le` is the usual order relation over real numbers. Notice that division is covariant in its first argument and contravariant in its second argument.

Leibniz equality is a relation and every function is a morphism that respects Leibniz equality. Unfortunately, Leibniz equality is not always the intended equality for a given structure.

In the next section we will describe the commands to register terms as parametric relations and morphisms. Several tactics that deal with equality in Coq can also work with the registered relations. The exact list of tactics will be given *in this section*. For instance, the tactic reflexivity can be used to solve a goal `R n n` whenever `R` is an instance of a registered reflexive relation. However, the tactics that replace in a context `C[]` one term with another one related by `R` must verify that `C[]` is a morphism that respects the intended relation. Currently the verification consists of checking whether `C[]` is a syntactic composition of morphism instances that respects some obvious compatibility constraints.

Example: Rewriting

Continuing the previous examples, suppose that the user must prove `set_eq int (union int (union int S1 S2) S2) (f S1 S2)` under the hypothesis `H : set_eq int S2 (@nil int)`. It is possible to use the `rewrite` tactic to replace the first two occurrences of `S2` with `@nil int` in the goal since the context `set_eq int (union int (union int S1 nil) nil) (f S1 S2)`, being a composition of morphisms instances, is a morphism. However the tactic will fail replacing the third occurrence of `S2` unless `f` has also been declared as a morphism.

Adding new relations and morphisms

Command: `Add Parametric Relation` *binders* `: (A t1 ... tn) (Aeq t 1 ... t m)` `reflexivity proved by` *term*

This command declares a parametric relation `Aeq: forall (y1 : β1 ... ym : βm)`, `relation (A t1 ... tn)` over `(A : αi -> ... αn -> Type)`.

The final *ident* gives a unique name to the morphism and it is used by the command to generate fresh names for automatically provided lemmas used internally.

Notice that the carrier and relation parameters may refer to the context of variables introduced at the beginning of the declaration, but the instances need not be made only of variables. Also notice that `A` is *not* required to be a term having the same parameters as `Aeq`, although that is often the case in practice (this departs from the previous implementation).

To use this command, you need to first import the module `Setoid` using the command `Require Import Setoid`.

Command: `Add Relation`

In case the carrier and relations are not parametric, one can use this command instead, whose syntax is the same except there is no local context.

The proofs of reflexivity, symmetry and transitivity can be omitted if the relation is not an equivalence relation. The proofs must be instances of the corresponding relation definitions: e.g. the proof of reflexivity must have a type convertible to `reflexive (A t1 ... tn) (Aeq t 1 …t n)`. Each proof may refer to the introduced variables as well.

Example: Parametric relation

For Leibniz equality, we may declare:

```
Add Parametric Relation (A : Type) : A (@eq A)
  [reflexivity proved by @refl_equal A]
...
```

Some tactics (*reflexivity*, *symmetry*, *transitivity*) work only on relations that respect the expected properties. The remaining tactics (*replace*, *rewrite* and derived tactics such as *autorewrite*) do not require any properties over the relation. However, they are able to replace terms with related ones only in contexts that are syntactic compositions of parametric morphism instances declared with the following command.

Command: `Add Parametric Morphism` *binders* `: (`*ident* *term*$_1$+`) with signature` *term*$_2$ `as` *ident*

This command declares a parametric morphism *ident* *term*$_1$+ of signature *term*$_2$. The final identifier *ident* gives a unique name to the morphism and it is used as the base name of the typeclass instance definition and as the name of the lemma that proves the well-definedness of the morphism. The parameters of the morphism as well as the signature may refer to the context of variables. The command asks the user to prove interactively that the function denoted by the first *ident* respects the relations identified from the signature.

Example

We start the example by assuming a small theory over homogeneous sets and we declare set equality as a parametric equivalence relation and union of two sets as a parametric morphism.

```
Require Export Setoid.
Require Export Relation_Definitions.
Set Implicit Arguments.
Parameter set : Type -> Type.
Parameter empty : forall A, set A.
Parameter eq_set : forall A, set A -> set A -> Prop.
Parameter union : forall A, set A -> set A -> set A.
Axiom eq_set_refl : forall A, reflexive _ (eq_set (A:=A)).
Axiom eq_set_sym : forall A, symmetric _ (eq_set (A:=A)).
Axiom eq_set_trans : forall A, transitive _ (eq_set (A:=A)).
Axiom empty_neutral : forall A (S : set A), eq_set (union S (empty A)) S.
Axiom union_compat :
  forall (A : Type),
    forall x x' : set A, eq_set x x' ->
    forall y y' : set A, eq_set y y' ->
      eq_set (union x y) (union x' y').
Add Parametric Relation A : (set A) (@eq_set A)
  reflexivity proved by (eq_set_refl (A:=A))
  symmetry proved by (eq_set_sym (A:=A))
  transitivity proved by (eq_set_trans (A:=A))
  as eq_set_rel.
Add Parametric Morphism A : (@union A)
  with signature (@eq_set A) ==> (@eq_set A) ==> (@eq_set A) as union_mor.
Proof.
exact (@union_compat A).
Qed.
```

It is possible to reduce the burden of specifying parameters using (maximally inserted) implicit arguments. If `A` is always set as maximally implicit in the previous example, one can write:

```
Add Parametric Relation A : (set A) eq_set
  reflexivity proved by eq_set_refl
  symmetry proved by eq_set_sym
  transitivity proved by eq_set_trans
  as eq_set_rel.

Add Parametric Morphism A : (@union A) with
  signature eq_set ==> eq_set ==> eq_set as union_mor.
Proof. exact (@union_compat A). Qed.
```

We proceed now by proving a simple lemma performing a rewrite step and then applying reflexivity, as we would do working with Leibniz equality. Both tactic applications are accepted since the required properties over `eq_set` and `union` can be established from the two declarations above.

```
Goal forall (S : set nat),
  eq_set (union (union S (empty nat)) S) (union S S).


Proof.
intros.
rewrite empty_neutral.
reflexivity.
Qed.
```

The tables of relations and morphisms are managed by the typeclass instance mechanism. The behavior on section close is to generalize the instances by the variables of the section (and possibly hypotheses used in the proofs of instance declarations) but not to export them in the rest of the development for proof search. One can use the cmd:`Existing Instance` command to do so outside the section, using the name of the declared morphism suffixed by `_Morphism`, or use the `Global` modifier for the corresponding class instance declaration (see *First Class Setoids and Morphisms*) at definition time. When loading a compiled file or importing a module, all the declarations of this module will be loaded.

Rewriting and non reflexive relations

To replace only one argument of an n-ary morphism it is necessary to prove that all the other arguments are related to themselves by the respective relation instances.

Example

To replace `(union S empty)` with `S` in `(union (union S empty) S) (union S S)` the rewrite tactic must exploit the monotony of `union` (axiom `union_compat` in the previous example). Applying `union_compat` by hand we are left with the goal `eq_set (union S S) (union S S)`.

When the relations associated to some arguments are not reflexive, the tactic cannot automatically prove the reflexivity goals, that are left to the user.

Setoids whose relations are partial equivalence relations (PER) are useful for dealing with partial functions. Let `R` be a PER. We say that an element `x` is defined if `R x x`. A partial function whose domain comprises all the defined elements is declared as a morphism that respects `R`. Every time a rewriting step is performed the user must prove that the argument of the morphism is defined.

Example

Let `eq0` be `fun x y => x = y /\ x <> 0` (the smallest PER over nonzero elements). Division can be declared as a morphism of signature `eq ==> eq0 ==> eq`. Replacing `x` with `y` in `div x n = div y n` opens an additional goal `eq0 n n` which is equivalent to `n = n /\ n <> 0`.

Rewriting and non symmetric relations

When the user works up to relations that are not symmetric, it is no longer the case that any covariant morphism argument is also contravariant. As a result it is no longer possible to replace a term with a related one in every context, since the obtained goal implies the previous one if and only if the replacement has been performed in a contravariant position. In a similar way, replacement in an hypothesis can be performed only if the replaced term occurs in a covariant position.

Example: Covariance and contravariance

Suppose that division over real numbers has been defined as a morphism of signature `Z.div : Z.lt ++> Z.lt --> Z.lt` (i.e. `Z.div` is increasing in its first argument, but decreasing on the second one). Let `<` denote `Z.lt`. Under the hypothesis `H : x < y` we have `k < x / y -> k < x / x`, but not `k < y / x -> k < x / x`. Dually, under the same hypothesis `k < x / y -> k < y / y` holds, but `k < y / x -> k < y / y` does not. Thus, if the current goal is `k < x / x`, it is possible to replace only the second occurrence of `x` (in contravariant position) with `y` since the obtained goal must imply the current one. On the contrary, if `k < x / x` is an hypothesis, it is possible to replace only the first occurrence of `x` (in covariant position) with `y` since the current hypothesis must imply the obtained one.

Contrary to the previous implementation, no specific error message will be raised when trying to replace a term that occurs in the wrong position. It will only fail because the rewriting constraints are not satisfiable. However it is possible to use the at modifier to specify which occurrences should be rewritten.

As expected, composing morphisms together propagates the variance annotations by switching the variance every time a contravariant position is traversed.

Example

Let us continue the previous example and let us consider the goal `x / (x / x) < k`. The first and third occurrences of `x` are in a contravariant position, while the second one is in covariant position. More in detail, the second occurrence of `x` occurs covariantly in `(x / x)` (since division is covariant in its first argument), and thus contravariantly in `x / (x / x)` (since division is contravariant in its second argument), and finally covariantly in `x / (x / x) < k` (since `<`, as every transitive relation, is contravariant in its first argument with respect to the relation itself).

Rewriting in ambiguous setoid contexts

One function can respect several different relations and thus it can be declared as a morphism having multiple signatures.

Example

Union over homogeneous lists can be given all the following signatures: `eq ==> eq ==> eq` (`eq` being the equality over ordered lists) `set_eq ==> set_eq ==> set_eq` (`set_eq` being the equality over unordered lists up to duplicates), `multiset_eq ==> multiset_eq ==> multiset_eq` (`multiset_eq` being the equality over unordered lists).

To declare multiple signatures for a morphism, repeat the *`Add Morphism`* command.

When morphisms have multiple signatures it can be the case that a rewrite request is ambiguous, since it is unclear what relations should be used to perform the rewriting. Contrary to the previous implementation, the tactic will always choose the first possible solution to the set of constraints generated by a rewrite and will not try to find *all* the possible solutions to warn the user about them.

8.11.2 Commands and tactics

First class setoids and morphisms

The implementation is based on a first-class representation of properties of relations and morphisms as typeclasses. That is, the various combinations of properties on relations and morphisms are represented as records and instances of these classes are put in a hint database. For example, the declaration:

```
Add Parametric Relation (x1 : T1) ... (xn : Tn) : (A t1 ... tn) (Aeq t 1 ... t m)
  [reflexivity proved by refl]
  [symmetry proved by sym]
  [transitivity proved by trans]
  as id.
```

is equivalent to an instance declaration:

```
Instance (x1 : T1) ... (xn : Tn) => id : @Equivalence (A t1 ... tn) (Aeq t 1 ... t m) :=
  [Equivalence_Reflexive := refl]
  [Equivalence_Symmetric := sym]
  [Equivalence_Transitive := trans].
```

The declaration itself amounts to the definition of an object of the record type `Coq.Classes.RelationClasses.Equivalence` and a hint added to the `typeclass_instances` hint database. Morphism declarations are also instances of a typeclass defined in `Classes.Morphisms`. See the documentation on *Typeclasses* and the theories files in Classes for further explanations.

One can inform the rewrite tactic about morphisms and relations just by using the typeclass mechanism to declare them using Instance and Context vernacular commands. Any object of type Proper (the type of morphism declarations) in the local context will also be automatically used by the rewriting tactic to solve constraints.

Other representations of first class setoids and morphisms can also be handled by encoding them as records. In the following example, the projections of the setoid relation and of the morphism function can be registered as parametric relations and morphisms.

Example: First class setoids

```
Require Import Relation_Definitions Setoid.
Record Setoid : Type :=
{ car: Type;
  eq: car -> car -> Prop;
  refl: reflexive _ eq;
  sym: symmetric _ eq;
  trans: transitive _ eq
}.
Add Parametric Relation (s : Setoid) : (@car s) (@eq s)
  reflexivity proved by (refl s)
  symmetry proved by (sym s)
  transitivity proved by (trans s) as eq_rel.
Record Morphism (S1 S2 : Setoid) : Type :=
{ f: car S1 -> car S2;
  compat: forall (x1 x2 : car S1), eq S1 x1 x2 -> eq S2 (f x1) (f x2)
}.
Add Parametric Morphism (S1 S2 : Setoid) (M : Morphism S1 S2) :
  (@f S1 S2 M) with signature (@eq S1 ==> @eq S2) as apply_mor.
Proof.
apply (compat S1 S2 M).
Qed.
Lemma test : forall (S1 S2 : Setoid) (m : Morphism S1 S2)
  (x y : car S1), eq S1 x y -> eq S2 (f _ _ m x) (f _ _ m y).
Proof.
intros.
rewrite H.
reflexivity.
Qed.
```

Tactics enabled on user provided relations

The following tactics, all prefixed by `setoid_`, deal with arbitrary registered relations and morphisms. Moreover, all the corresponding unprefixed tactics (i.e. *`reflexivity`*, *`symmetry`*, *`transitivity`*, *`replace`*, *`rewrite`*) have been extended to fall back to their prefixed counterparts when the relation involved is not Leibniz equality. Notice, however, that using the prefixed tactics it is possible to pass additional arguments such as `using relation`.

Variant: `setoid_reflexivity`

Variant: `setoid_symmetry` `in` *`ident`*?

Variant: `setoid_transitivity`

Variant: `setoid_rewrite` [*orientation*]? *term* [`at` *occurrences*]? [`in` *ident*]?

Variant: `setoid_replace` *term* `with` *term* [`using relation` *term*]? [`in` *ident*]? [`by` *tactic*]?

The `using relation` arguments cannot be passed to the unprefixed form. The latter argument tells the tactic what parametric relation should be used to replace the first tactic argument with the second one. If omitted, it defaults to the `DefaultRelation` instance on the type of the objects. By default, it means the most recent `Equivalence` instance in the environment, but it can be customized by declaring new `DefaultRelation` instances. As Leibniz equality is a declared equivalence, it will fall back to it if no other relation is declared on a given type.

Every derived tactic that is based on the unprefixed forms of the tactics considered above will also work up to user defined relations. For instance, it is possible to register hints for *autorewrite* that are not proofs of Leibniz equalities. In particular it is possible to exploit *autorewrite* to simulate normalization in a term rewriting system up to user defined equalities.

Printing relations and morphisms

Command: `Print Instances`

This command can be used to show the list of currently registered `Reflexive` (using `Print Instances Reflexive`), `Symmetric` or `Transitive` relations, Equivalences, PreOrders, PERs, and Morphisms (implemented as `Proper` instances). When the rewriting tactics refuse to replace a term in a context because the latter is not a composition of morphisms, the *Print Instances* command can be useful to understand what additional morphisms should be registered.

Deprecated syntax and backward incompatibilities

Command: `Add Setoid` *qualid*$_1$ *qualid*$_2$ *qualid*$_3$ `as` *ident*

This command for declaring setoids and morphisms is also accepted due to backward compatibility reasons.

Here *qualid*$_2$ is a congruence relation without parameters, *qualid*$_1$ is its carrier and *qualid*$_3$ is an object of type (`Setoid_Theory` *qualid*$_1$ *qualid*$_2$) (i.e. a record packing together the reflexivity, symmetry and transitivity lemmas). Notice that the syntax is not completely backward compatible since the identifier was not required.

Command: `Add Morphism` *ident* `:` *ident*

This command is restricted to the declaration of morphisms without parameters. It is not fully backward compatible since the property the user is asked to prove is slightly different: for n-ary morphisms the hypotheses of the property are permuted; moreover, when the morphism returns a proposition, the property is now stated using a bi-implication in place of a simple implication. In practice, porting an old development to the new semantics is usually quite simple.

Command: `Declare Morphism` *ident* `:` *ident*

This commands is to be used in a module type to declare a parameter that is a morphism.

Notice that several limitations of the old implementation have been lifted. In particular, it is now possible to declare several relations with the same carrier and several signatures for the same morphism. Moreover, it is now also possible to declare several morphisms having the same signature. Finally, the *replace* and *rewrite* tactics can be used to replace terms in contexts that were refused by the old implementation. As discussed in the next section, the semantics of the new *setoid_rewrite* tactic differs slightly from the old one and *rewrite*.

8.11.3 Extensions

Rewriting under binders

Warning: Due to compatibility issues, this feature is enabled only when calling the *setoid_rewrite* tactic directly and not *rewrite*.

To be able to rewrite under binding constructs, one must declare morphisms with respect to pointwise (setoid) equivalence of functions. Example of such morphisms are the standard `all` and `ex` combinators for universal and existential quantification respectively. They are declared as morphisms in the `Classes.Morphisms_Prop` module. For example, to declare that universal quantification is a morphism for logical equivalence:

```
Instance all_iff_morphism (A : Type) :
        Proper (pointwise_relation A iff ==> iff) (@all A).


Proof.
simpl_relation.
    1 subgoal

      A : Type
      x, y : A -> Prop
      H : pointwise_relation A iff x y
      ============================
      all x <-> all y
```

One then has to show that if two predicates are equivalent at every point, their universal quantifications are equivalent. Once we have declared such a morphism, it will be used by the setoid rewriting tactic each time we try to rewrite under an `all` application (products in `Prop` are implicitly translated to such applications).

Indeed, when rewriting under a lambda, binding variable `x`, say from `P x` to `Q x` using the relation iff, the tactic will generate a proof of `pointwise_relation A iff (fun x => P x) (fun x => Q x)` from the proof of `iff (P x) (Q x)` and a constraint of the form `Proper (pointwise_relation A iff ==> ?) m` will be generated for the surrounding morphism `m`.

Hence, one can add higher-order combinators as morphisms by providing signatures using pointwise extension for the relations on the functional arguments (or whatever subrelation of the pointwise extension). For example, one could declare the `map` combinator on lists as a morphism:

```
Instance map_morphism `{Equivalence A eqA, Equivalence B eqB} :
        Proper ((eqA ==> eqB) ==> list_equiv eqA ==> list_equiv eqB) (@map A B).
```

where `list_equiv` implements an equivalence on lists parameterized by an equivalence on the elements.

Note that when one does rewriting with a lemma under a binder using *setoid_rewrite*, the application of the lemma may capture the bound variable, as the semantics are different from rewrite where the lemma is first matched on the whole term. With the new *setoid_rewrite*, matching is done on each subterm separately and in its local environment, and all matches are rewritten *simultaneously* by default. The semantics of the previous *setoid_rewrite* implementation can almost be recovered using the `at 1` modifier.

Subrelations

Subrelations can be used to specify that one relation is included in another, so that morphism signatures for one can be used for the other. If a signature mentions a relation `R` on the left of an arrow `==>`, then the signature also applies for any relation `S` that is smaller than `R`, and the inverse applies on the right of an arrow. One can then declare only a few morphisms instances that generate the complete set of signatures for a particular constant. By default, the only declared subrelation is `iff`, which is a subrelation of `impl` and `inverse impl` (the dual of implication). That's why we can declare only two morphisms for conjunction: `Proper (impl ==> impl ==> impl) and` and `Proper (iff ==> iff ==> iff) and`. This is sufficient to satisfy any rewriting constraints arising from a rewrite using `iff`, `impl` or `inverse impl` through `and`.

Subrelations are implemented in `Classes.Morphisms` and are a prime example of a mostly user-space extension of the algorithm.

Constant unfolding

The resolution tactic is based on typeclasses and hence regards user- defined constants as transparent by default. This may slow down the resolution due to a lot of unifications (all the declared `Proper` instances are tried at each node of the search tree). To speed it up, declare your constant as rigid for proof search using the command *`Typeclasses Opaque`*.

8.11.4 Strategies for rewriting

Definitions

The generalized rewriting tactic is based on a set of strategies that can be combined to obtain custom rewriting procedures. Its set of strategies is based on the programmable rewriting strategies with generic traversals by Visser et al. *[LV97] [VBT98]*, which formed the core of the Stratego transformation language *[Vis01]*. Rewriting strategies are applied using the tactic `rewrite_strat` *`strategy`* where *`strategy`* is a strategy expression. Strategies are defined inductively as described by the following grammar:

```
strategy  ::=  qualid (lemma, left to right)
               <- qualid (lemma, right to left)
               fail (failure)
               id (identity)
               refl (reflexivity)
               progress strategy (progress)
               try strategy (try catch)
               strategy ; strategy (composition)
               choice strategy strategy (left_biased_choice)
               repeat strategy (one or more)
               any strategy (zero or more)
               subterm strategy (one subterm)
               subterms strategy (all subterms)
               innermost strategy (innermost first)
               outermost strategy (outermost first)
               bottomup strategy (bottom-up)
               topdown strategy (top-down)
               hints ident (apply hints from hint database)
               terms term ... term (any of the terms)
               eval redexpr (apply reduction)
               fold term (unify)
               ( strategy )
```

Actually a few of these are defined in term of the others using a primitive fixpoint operator:

- `try strategy := choice strategy id`

- `any` *strategy* `:= fix` *ident*`. try (`*strategy* `;` *ident*`)`
- `repeat` *strategy* `:=` *strategy*`; any` *strategy*
- `bottomup` *strategy* `:= fix` *ident*`. (choice (progress (subterms` *ident*`))` *strategy*`) ; try` *ident*
- `topdown` *strategy* `:= fix` *ident*`. (choice` *strategy* `(progress (subterms` *ident*`))) ; try` *ident*
- `innermost` *strategy* `:= fix` *ident*`. (choice (subterm` *ident*`)` *strategy*`)`
- `outermost` *strategy* `:= fix` *ident*`. (choice` *strategy* `(subterm` *ident*`))`

The basic control strategy semantics are straightforward: strategies are applied to subterms of the term to rewrite, starting from the root of the term. The lemma strategies unify the left-hand-side of the lemma with the current subterm and on success rewrite it to the right- hand-side. Composition can be used to continue rewriting on the current subterm. The `fail` strategy always fails while the identity strategy succeeds without making progress. The reflexivity strategy succeeds, making progress using a reflexivity proof of rewriting. `progress` tests progress of the argument *strategy* and fails if no progress was made, while `try` always succeeds, catching failures. `choice` is left-biased: it will launch the first strategy and fall back on the second one in case of failure. One can iterate a strategy at least 1 time using `repeat` and at least 0 times using `any`.

The `subterm` and `subterms` strategies apply their argument *strategy* to respectively one or all subterms of the current term under consideration, left-to-right. `subterm` stops at the first subterm for which *strategy* made progress. The composite strategies `innermost` and `outermost` perform a single innermost or outermost rewrite using their argument *strategy*. Their counterparts `bottomup` and `topdown` perform as many rewritings as possible, starting from the bottom or the top of the term.

Hint databases created for *autorewrite* can also be used by *rewrite_strat* using the `hints` strategy that applies any of the lemmas at the current subterm. The `terms` strategy takes the lemma names directly as arguments. The `eval` strategy expects a reduction expression (see *Performing computations*) and succeeds if it reduces the subterm under consideration. The `fold` strategy takes a *term* and tries to *unify* it to the current subterm, converting it to *term* on success. It is stronger than the tactic `fold`.

Usage

`rewrite_strat` *strategy* `in` *ident*?

Rewrite using the strategy s in hypothesis ident or the conclusion.

Error: `Nothing to rewrite.`

If the strategy failed.

Error: `No progress made.`

If the strategy succeeded but made no progress.

Error: `Unable to satisfy the rewriting constraints.`

If the strategy succeeded and made progress but the corresponding rewriting constraints are not satisfied.

The `setoid_rewrite c` tactic is basically equivalent to `rewrite_strat (outermost c)`.

8.12 Asynchronous and Parallel Proof Processing

Author Enrico Tassi

This chapter explains how proofs can be asynchronously processed by Coq. This feature improves the reactivity of the system when used in interactive mode via CoqIDE. In addition, it allows Coq to take advantage of parallel hardware when used as a batch compiler by decoupling the checking of statements and definitions from the construction and checking of proofs objects.

This feature is designed to help dealing with huge libraries of theorems characterized by long proofs. In the current state, it may not be beneficial on small sets of short files.

This feature has some technical limitations that may make it unsuitable for some use cases.

For example, in interactive mode, some errors coming from the kernel of Coq are signaled late. The type of errors belonging to this category are universe inconsistencies.

At the time of writing, only opaque proofs (ending with `Qed` or `Admitted`) can be processed asynchronously.

Finally, asynchronous processing is disabled when running CoqIDE in Windows. The current implementation of the feature is not stable on Windows. It can be enabled, as described below at *Interactive mode*, though doing so is not recommended.

8.12.1 Proof annotations

To process a proof asynchronously Coq needs to know the precise statement of the theorem without looking at the proof. This requires some annotations if the theorem is proved inside a Section (see Section *Section mechanism*).

When a section ends, Coq looks at the proof object to decide which section variables are actually used and hence have to be quantified in the statement of the theorem. To avoid making the construction of proofs mandatory when ending a section, one can start each proof with the `Proof using` command (Section *Switching on/off the proof editing mode*) that declares which section variables the theorem uses.

The presence of `Proof` using is needed to process proofs asynchronously in interactive mode.

It is not strictly mandatory in batch mode if it is not the first time the file is compiled and if the file itself did not change. When the proof does not begin with Proof using, the system records in an auxiliary file, produced along with the `.vo` file, the list of section variables used.

Automatic suggestion of proof annotations

The *`Suggest Proof Using`* flag makes Coq suggest, when a `Qed` command is processed, a correct proof annotation. It is up to the user to modify the proof script accordingly.

8.12.2 Proof blocks and error resilience

Coq 8.6 introduced a mechanism for error resilience: in interactive mode Coq is able to completely check a document containing errors instead of bailing out at the first failure.

Two kind of errors are supported: errors occurring in vernacular commands and errors occurring in proofs.

To properly recover from a failing tactic, Coq needs to recognize the structure of the proof in order to confine the error to a sub proof. Proof block detection is performed by looking at the syntax of the proof script (i.e. also looking at indentation). Coq comes with four kind of proof blocks, and an ML API to add new ones.

curly blocks are delimited by { and }, see Chapter *Proof handling*

par blocks are atomic, i.e. just one tactic introduced by the `par:` goal selector

indent blocks end with a tactic indented less than the previous one

bullet blocks are delimited by two equal bullet signs at the same indentation level

Caveats

When a vernacular command fails the subsequent error messages may be bogus, i.e. caused by the first error. Error resilience for vernacular commands can be switched off by passing `-async-proofs-command-error-resilience off` to CoqIDE.

An incorrect proof block detection can result into an incorrect error recovery and hence in bogus errors. Proof block detection cannot be precise for bullets or any other non well parenthesized proof structure. Error resilience can be turned off or selectively activated for any set of block kind passing to CoqIDE one of the following options:

- `-async-proofs-tactic-error-resilience off`
- `-async-proofs-tactic-error-resilience all`
- `-async-proofs-tactic-error-resilience` $\boxed{\texttt{blocktype}}^{*}_{,}$

Valid proof block types are: “curly”, “par”, “indent”, and “bullet”.

8.12.3 Interactive mode

At the time of writing the only user interface supporting asynchronous proof processing is CoqIDE.

When CoqIDE is started, two Coq processes are created. The master one follows the user, giving feedback as soon as possible by skipping proofs, which are delegated to the worker process. The worker process, whose state can be seen by clicking on the button in the lower right corner of the main CoqIDE window, asynchronously processes the proofs. If a proof contains an error, it is reported in red in the label of the very same button, that can also be used to see the list of errors and jump to the corresponding line.

If a proof is processed asynchronously the corresponding Qed command is colored using a lighter color than usual. This signals that the proof has been delegated to a worker process (or will be processed lazily if the `-async-proofs lazy` option is used). Once finished, the worker process will provide the proof object, but this will not be automatically checked by the kernel of the main process. To force the kernel to check all the proof objects, one has to click the button with the gears (Fully check the document) on the top bar. Only then all the universe constraints are checked.

Caveats

The number of worker processes can be increased by passing CoqIDE the `-async-proofs-j n` flag. Note that the memory consumption increases too, since each worker requires the same amount of memory as the master process. Also note that increasing the number of workers may reduce the reactivity of the master process to user commands.

To disable this feature, one can pass the `-async-proofs off` flag to CoqIDE. Conversely, on Windows, where the feature is disabled by default, pass the `-async-proofs on` flag to enable it.

Proofs that are known to take little time to process are not delegated to a worker process. The threshold can be configured with `-async-proofs-delegation-threshold`. Default is 0.03 seconds.

8.12.4 Batch mode

Warning: The `-vio` flag is subsumed, for most practical usage, by the the more recent `-vos` flag. See *Compiled interfaces (produced using -vos)*.

Warning: When working with `.vio` files, do not use the `-vos` option at the same time, otherwise stale files might get loaded when executing a `Require`. Indeed, the loading of a nonempty `.vos` file is assigned higher priority than the loading of a `.vio` file.

When Coq is used as a batch compiler by running `coqc`, it produces a `.vo` file for each `.v` file. A `.vo` file contains, among other things, theorem statements and proofs. Hence to produce a .vo Coq need to process all the proofs of the `.v` file.

The asynchronous processing of proofs can decouple the generation of a compiled file (like the `.vo` one) that can be loaded by `Require` from the generation and checking of the proof objects. The `-vio` flag can be passed to `coqc` to produce, quickly, `.vio` files. Alternatively, when using a Makefile produced by `coq_makefile`, the `vio` target can be used to compile all files using the `-vio` flag.

A `.vio` file can be loaded using `Require` exactly as a `.vo` file but proofs will not be available (the Print command produces an error). Moreover, some universe constraints might be missing, so universes inconsistencies might go unnoticed. A `.vio` file does not contain proof objects, but proof tasks, i.e. what a worker process can transform into a proof object.

Compiling a set of files with the `-vio` flag allows one to work, interactively, on any file without waiting for all the proofs to be checked.

When working interactively, one can fully check all the `.v` files by running `coqc` as usual.

Alternatively one can turn each `.vio` into the corresponding `.vo`. All .vio files can be processed in parallel, hence this alternative might be faster. The command `coqc -schedule-vio2vo 2 a b c` can be used to obtain a good scheduling for two workers to produce `a.vo`, `b.vo`, and `c.vo`. When using a Makefile produced by `coq_makefile`, the `vio2vo` target can be used for that purpose. Variable `J` should be set to the number of workers, e.g. `make vio2vo J=2`. The only caveat is that, while the .vo files obtained from `.vio` files are complete (they contain all proof terms and universe constraints), the satisfiability of all universe constraints has not been checked globally (they are checked to be consistent for every single proof). Constraints will be checked when these `.vo` files are (recursively) loaded with `Require`.

There is an extra, possibly even faster, alternative: just check the proof tasks stored in `.vio` files without producing the `.vo` files. This is possibly faster because all the proof tasks are independent, hence one can further partition the job to be done between workers. The `coqc -schedule-vio-checking 6 a b c` command can be used to obtain a good scheduling for 6 workers to check all the proof tasks of `a.vio`, `b.vio`, and `c.vio`. Auxiliary files are used to predict how long a proof task will take, assuming it will take the same amount of time it took last time. When using a Makefile produced by coq_makefile, the `checkproofs` target can be used to check all `.vio` files. Variable `J` should be set to the number of workers, e.g. `make checkproofs J=6`. As when converting `.vio` files to `.vo` files, universe constraints are not checked to be globally consistent. Hence this compilation mode is only useful for quick regression testing and on developments not making heavy use of the `Type` hierarchy.

8.12.5 Limiting the number of parallel workers

Many Coq processes may run on the same computer, and each of them may start many additional worker processes. The `coqworkmgr` utility lets one limit the number of workers, globally.

The utility accepts the `-j` argument to specify the maximum number of workers (defaults to 2). `coqworkmgr` automatically starts in the background and prints an environment variable assignment like

COQWORKMGR_SOCKET=localhost:45634. The user must set this variable in all the shells from which Coq processes will be started. If one uses just one terminal running the bash shell, then export ‘coqworkmgr -j 4‘ will do the job.

After that, all Coq processes, e.g. coqide and coqc, will respect the limit, globally.

8.13 Miscellaneous extensions

8.13.1 Program derivation

Coq comes with an extension called Derive, which supports program derivation. Typically in the style of Bird and Meertens or derivations of program refinements. To use the Derive extension it must first be required with Require Coq.derive.Derive. When the extension is loaded, it provides the following command:

Command: **Derive** *ident₁* **SuchThat** *type* **As** *ident₂*

ident₁ can appear in *type*. This command opens a new proof presenting the user with a goal for *type* in which the name *ident₁* is bound to an existential variable ?x (formally, there are other goals standing for the existential variables but they are shelved, as described in *shelve*).

When the proof ends two constants are defined:

- The first one is named *ident₁* and is defined as the proof of the shelved goal (which is also the value of ?x). It is always transparent.
- The second one is named *ident₂*. It has type *type*, and its body is the proof of the initially visible goal. It is opaque if the proof ends with *Qed*, and transparent if the proof ends with *Defined*.

Example

```
Require Coq.derive.Derive.
    [Loading ML file derive_plugin.cmxs ... done]

Require Import Coq.Numbers.Natural.Peano.NPeano.
Section P.
Variables (n m k:nat).
    n is declared
    m is declared
    k is declared

Derive p SuchThat ((k*n)+(k*m) = p) As h.
    1 focused subgoal
    (shelved: 1)

      n, m, k : nat
      p := ?Goal : nat
      ============================
      k * n + k * m = p

Proof.
rewrite <- Nat.mul_add_distr_l.
    1 focused subgoal
    (shelved: 1)

      n, m, k : nat
```

(continues on next page)

(continued from previous page)

```
      p := ?Goal : nat
      ============================
      k * (n + m) = p

subst p.
    1 focused subgoal
    (shelved: 1)

      n, m, k : nat
      ============================
      k * (n + m) = ?Goal

reflexivity.
    No more subgoals.

Qed.
End P.
Print p.
    p = fun n m k : nat => k * (n + m)
         : nat -> nat -> nat -> nat

    Arguments p (_ _ _)%nat_scope

Check h.
    h
         : forall n m k : nat, k * n + k * m = p n m k
```

Any property can be used as `term`, not only an equation. In particular, it could be an order relation specifying some form of program refinement or a non-executable property from which deriving a program is convenient.

8.14 Polymorphic Universes

Author Matthieu Sozeau

8.14.1 General Presentation

> **Warning:** The status of Universe Polymorphism is experimental.

This section describes the universe polymorphic extension of Coq. Universe polymorphism makes it possible to write generic definitions making use of universes and reuse them at different and sometimes incompatible universe levels.

A standard example of the difference between universe *polymorphic* and *monomorphic* definitions is given by the identity function:

```
Definition identity {A : Type} (a : A) := a.
```

By default, constant declarations are monomorphic, hence the identity function declares a global universe (say `Top.1`) for its domain. Subsequently, if we try to self-apply the identity, we will get an error:

```
Fail Definition selfid := identity (@identity).
    The command has indeed failed with message:
    The term "@identity" has type "forall A : Type, A -> A"
    while it is expected to have type "?A"
    (unable to find a well-typed instantiation for "?A": cannot ensure that
    "Type@{identity.u0+1}" is a subtype of "Type@{identity.u0}").
```

Indeed, the global level `Top.1` would have to be strictly smaller than itself for this self-application to type check, as the type of `(@identity)` is `forall (A : Type@{Top.1}), A -> A` whose type is itself `Type@{Top.1+1}`.

A universe polymorphic identity function binds its domain universe level at the definition level instead of making it global.

```
Polymorphic Definition pidentity {A : Type} (a : A) := a.

About pidentity.
    pidentity@{Top.2} : forall A : Type, A -> A

    pidentity is universe polymorphic
    Arguments pidentity {A}%type_scope
    pidentity is transparent
    Expands to: Constant Top.pidentity
```

It is then possible to reuse the constant at different levels, like so:

```
Definition selfpid := pidentity (@pidentity).
```

Of course, the two instances of `pidentity` in this definition are different. This can be seen when the *Printing Universes* flag is on:

```
Print selfpid.
    selfpid =
    pidentity@{selfpid.u0} (@pidentity@{selfpid.u1})
         : forall A : Type@{selfpid.u1}, A -> A
    (* {selfpid.u1 selfpid.u0} |= selfpid.u1 < selfpid.u0 *)

    Arguments selfpid _%type_scope
```

Now `pidentity` is used at two different levels: at the head of the application it is instantiated at `Top.3` while in the argument position it is instantiated at `Top.4`. This definition is only valid as long as `Top.4` is strictly smaller than `Top.3`, as shown by the constraints. Note that this definition is monomorphic (not universe polymorphic), so the two universes (in this case `Top.3` and `Top.4`) are actually global levels.

When printing `pidentity`, we can see the universes it binds in the annotation `@{Top.2}`. Additionally, when *Printing Universes* is on we print the ”universe context” of `pidentity` consisting of the bound universes and the constraints they must verify (for `pidentity` there are no constraints).

Inductive types can also be declared universes polymorphic on universes appearing in their parameters or fields. A typical example is given by monoids:

```
Polymorphic Record Monoid := { mon_car :> Type; mon_unit : mon_car;
  mon_op : mon_car -> mon_car -> mon_car }.

Print Monoid.
```

The Monoid’s carrier universe is polymorphic, hence it is possible to instantiate it for example with `Monoid` itself. First we build the trivial unit monoid in `Set`:

```
Definition unit_monoid : Monoid :=
  {| mon_car := unit; mon_unit := tt; mon_op x y := tt |}.
```

From this we can build a definition for the monoid of Set-monoids (where multiplication would be given by the product of monoids).

```
Polymorphic Definition monoid_monoid : Monoid.
refine (@Build_Monoid Monoid unit_monoid (fun x y => x)).
Defined.

Print monoid_monoid.
    monoid_monoid@{Top.9} =
    {|
    mon_car := Monoid@{Set};
    mon_unit := unit_monoid;
    mon_op := fun x _ : Monoid@{Set} => x |}
         : Monoid@{Top.9}
    (* Top.9 |= Set < Top.9 *)
```

As one can see from the constraints, this monoid is "large", it lives in a universe strictly higher than Set.

8.14.2 Polymorphic, Monomorphic

Command: `Polymorphic` *definition*

As shown in the examples, polymorphic definitions and inductives can be declared using the Polymorphic prefix.

Flag: `Universe Polymorphism`

Once enabled, this flag will implicitly prepend Polymorphic to any definition of the user.

Command: `Monomorphic` *definition*

When the *Universe Polymorphism* flag is set, to make a definition producing global universe constraints, one can use the Monomorphic prefix.

Many other commands support the Polymorphic flag, including:

- Lemma, Axiom, and all the other "definition" keywords support polymorphism.
- *Section* will locally set the polymorphism flag inside the section.
- Variables, Context, Universe and Constraint in a section support polymorphism. See *Universe polymorphism and sections* for more details.
- *Hint Resolve* and *Hint Rewrite* will use the auto/rewrite hint polymorphically, not at a single instance.

8.14.3 Cumulative, NonCumulative

Polymorphic inductive types, coinductive types, variants and records can be declared cumulative using the Cumulative prefix.

Command: `Cumulative` *inductive*

Declares the inductive as cumulative

Alternatively, there is a *Polymorphic Inductive Cumulativity* flag which when set, makes all subsequent *polymorphic* inductive definitions cumulative. When set, inductive types and the like can be enforced to be non-cumulative using the NonCumulative prefix.

Command: `NonCumulative` *`inductive`*

Declares the inductive as non-cumulative

Flag: `Polymorphic Inductive Cumulativity`

When this flag is on, it sets all following polymorphic inductive types as cumulative (it is off by default).

Consider the examples below.

```
Polymorphic Cumulative Inductive list {A : Type} :=
| nil : list
| cons : A -> list -> list.


Print list.
    Inductive list@{Top.12} (A : Type@{Top.12}) : Type@{max(Set,Top.12)} :=
        nil : list@{Top.12} | cons : A -> list@{Top.12} -> list@{Top.12}
    (* *Top.12 |=  *)

    Arguments list {A}%type_scope
    Arguments nil {A}%type_scope
    Arguments cons {A}%type_scope
```

When printing `list`, the universe context indicates the subtyping constraints by prefixing the level names with symbols.

Because inductive subtypings are only produced by comparing inductives to themselves with universes changed, they amount to variance information: each universe is either invariant, covariant or irrelevant (there are no contravariant subtypings in Coq), respectively represented by the symbols `=`, `+` and `*`.

Here we see that `list` binds an irrelevant universe, so any two instances of `list` are convertible: $E[\Gamma] \vdash \mathsf{list@}\{i\}\ A =_{\beta\delta\iota\zeta\eta} \mathsf{list@}\{j\}\ B$ whenever $E[\Gamma] \vdash A =_{\beta\delta\iota\zeta\eta} B$ and this applies also to their corresponding constructors, when they are comparable at the same type.

See *Conversion rules* for more details on convertibility and subtyping. The following is an example of a record with non-trivial subtyping relation:

```
Polymorphic Cumulative Record packType := {pk : Type}.
    packType is defined
    pk is defined
```

`packType` binds a covariant universe, i.e.

$$E[\Gamma] \vdash \mathsf{packType@}\{i\} =_{\beta\delta\iota\zeta\eta} \mathsf{packType@}\{j\} \quad \text{whenever} \quad i \leq j$$

Cumulative inductive types, coinductive types, variants and records only make sense when they are universe polymorphic. Therefore, an error is issued whenever the user uses the `Cumulative` or `NonCumulative` prefix in a monomorphic context. Notice that this is not the case for the *`Polymorphic Inductive Cumulativity`* flag. That is, this flag, when set, makes all subsequent *polymorphic* inductive declarations cumulative (unless, of course the `NonCumulative` prefix is used) but has no effect on *monomorphic* inductive declarations.

Consider the following examples.

```
Fail Monomorphic Cumulative Inductive Unit := unit.
    The command has indeed failed with message:
    The Cumulative prefix can only be used in a polymorphic context.


Fail Monomorphic NonCumulative Inductive Unit := unit.
    The command has indeed failed with message:
    The NonCumulative prefix can only be used in a polymorphic context.
```

```
Set Polymorphic Inductive Cumulativity.
Inductive Unit := unit.
    Unit is defined
    Unit_rect is defined
    Unit_ind is defined
    Unit_rec is defined
    Unit_sind is defined
```

An example of a proof using cumulativity

```
Set Universe Polymorphism.
Set Polymorphic Inductive Cumulativity.
Inductive eq@{i} {A : Type@{i}} (x : A) : A -> Type@{i} := eq_refl : eq x x.
Definition funext_type@{a b e} (A : Type@{a}) (B : A -> Type@{b})
:= forall f g : (forall a, B a),
                (forall x, eq@{e} (f x) (g x))
                -> eq@{e} f g.
Section down.
Universes a b e e'.
Constraint e' < e.
Lemma funext_down {A B}
     (H : @funext_type@{a b e} A B) : @funext_type@{a b e'} A B.
Proof.
exact H.
Defined.
End down.
```

8.14.4 Cumulativity Weak Constraints

Flag: `Cumulativity Weak Constraints`

When set, which is the default, causes "weak" constraints to be produced when comparing universes in an irrelevant position. Processing weak constraints is delayed until minimization time. A weak constraint between `u` and `v` when neither is smaller than the other and one is flexible causes them to be unified. Otherwise the constraint is silently discarded.

This heuristic is experimental and may change in future versions. Disabling weak constraints is more predictable but may produce arbitrary numbers of universes.

8.14.5 Global and local universes

Each universe is declared in a global or local environment before it can be used. To ensure compatibility, every *global* universe is set to be strictly greater than `Set` when it is introduced, while every *local* (i.e. polymorphically quantified) universe is introduced as greater or equal to `Set`.

8.14.6 Conversion and unification

The semantics of conversion and unification have to be modified a little to account for the new universe instance arguments to polymorphic references. The semantics respect the fact that definitions are transparent, so indistinguishable from their bodies during conversion.

This is accomplished by changing one rule of unification, the first- order approximation rule, which applies when two applicative terms with the same head are compared. It tries to short-cut unfolding by comparing the arguments directly. In case the constant is universe polymorphic, we allow this rule to fire only when unifying the universes results in instantiating a so-called flexible universe variables (not given by the user). Similarly for conversion, if such an equation of applicative terms fail due to a universe comparison not being satisfied, the terms are unfolded. This change implies that conversion and unification can have different unfolding behaviors on the same development with universe polymorphism switched on or off.

8.14.7 Minimization

Universe polymorphism with cumulativity tends to generate many useless inclusion constraints in general. Typically at each application of a polymorphic constant `f`, if an argument has expected type `Type@{i}` and is given a term of type `Type@{j}`, a $j \leq i$ constraint will be generated. It is however often the case that an equation $j = i$ would be more appropriate, when `f`'s universes are fresh for example. Consider the following example:

```
Definition id0 := @pidentity nat 0.
```

```
Print id0.
    id0@{} = pidentity@{Set} 0
         : nat
```

This definition is elaborated by minimizing the universe of `id0` to level `Set` while the more general definition would keep the fresh level `i` generated at the application of `id` and a constraint that `Set` $\leq i$. This minimization process is applied only to fresh universe variables. It simply adds an equation between the variable and its lower bound if it is an atomic universe (i.e. not an algebraic max() universe).

Flag: `Universe Minimization ToSet`

Turning this flag off (it is on by default) disallows minimization to the sort `Set` and only collapses floating universes between themselves.

8.14.8 Explicit Universes

The syntax has been extended to allow users to explicitly bind names to universes and explicitly instantiate polymorphic definitions.

Command: `Universe` *`ident`*
Command: `Polymorphic Universe` *`ident`*

In the monorphic case, this command declares a new global universe named `ident`, which can be referred to using its qualified name as well. Global universe names live in a separate namespace. The command supports the `Polymorphic` flag only in sections, meaning the universe quantification will be discharged on each section definition independently.

Command: `Constraint` *`universe_constraint`*
Command: `Polymorphic Constraint` *`universe_constraint`*

This command declares a new constraint between named universes.

```
universe_constraint  ::=  qualid < qualid
                          qualid <= qualid
                          qualid = qualid
```

If consistent, the constraint is then enforced in the global environment. Like *`Universe`*, it can be used with the `Polymorphic` prefix in sections only to declare constraints discharged at section closing time. One cannot declare a global constraint on polymorphic universes.

Error: `Undeclared universe` *ident*.

Error: `Universe inconsistency.`

Polymorphic definitions

For polymorphic definitions, the declaration of (all) universe levels introduced by a definition uses the following syntax:

```
Polymorphic Definition le@{i j} (A : Type@{i}) : Type@{j} := A.
```

```
Print le.
    le@{i j} =
    fun A : Type@{i} => A
         : Type@{i} -> Type@{j}
    (* i j |= i <= j *)

    Arguments le _%type_scope
```

During refinement we find that `j` must be larger or equal than `i`, as we are using `A : Type@{i} <= Type@{j}`, hence the generated constraint. At the end of a definition or proof, we check that the only remaining universes are the ones declared. In the term and in general in proof mode, introduced universe names can be referred to in terms. Note that local universe names shadow global universe names. During a proof, one can use *Show Universes* to display the current context of universes.

It is possible to provide only some universe levels and let Coq infer the others by adding a `+` in the list of bound universe levels:

```
Fail Definition foobar@{u} : Type@{u} := Type.
    The command has indeed failed with message:
    Universe {Top.50} is unbound.
```

```
Definition foobar@{u +} : Type@{u} := Type.
    foobar is defined
```

```
Set Printing Universes.
Print foobar.
    foobar@{u Top.52} = Type@{Top.52}
         : Type@{u}
    (* u Top.52 |= Top.52 < u *)
```

This can be used to find which universes need to be explicitly bound in a given definition.

Definitions can also be instantiated explicitly, giving their full instance:

```
Check (pidentity@{Set}).
    pidentity@{Set}
         : ?A -> ?A
    where
    ?A : [ |- Set]
```

```
Monomorphic Universes k l.
Check (le@{k l}).
    le@{k l}
         : Type@{k} -> Type@{l}
    (* {} |= k <= l *)
```

User-named universes and the anonymous universe implicitly attached to an explicit `Type` are considered rigid for unification and are never minimized. Flexible anonymous universes can be produced with an underscore or by omitting the annotation to a polymorphic definition.

```
Check (fun x => x) : Type -> Type.
    (fun x : Type@{Top.55} => x) : Type@{Top.55} -> Type@{Top.56}
         : Type@{Top.55} -> Type@{Top.56}
    (* {Top.56 Top.55} |= Top.55 <= Top.56 *)

Check (fun x => x) : Type -> Type@{_}.
    (fun x : Type@{Top.57} => x) : Type@{Top.57} -> Type@{Top.57}
         : Type@{Top.57} -> Type@{Top.57}
    (* {Top.57} |=  *)

Check le@{k _}.
    le@{k k}
         : Type@{k} -> Type@{k}

Check le.
    le@{Top.60 Top.60}
         : Type@{Top.60} -> Type@{Top.60}
    (* {Top.60} |=  *)
```

Flag: `Strict Universe Declaration`

Turning this flag off allows one to freely use identifiers for universes without declaring them first, with the semantics that the first use declares it. In this mode, the universe names are not associated with the definition or proof once it has been defined. This is meant mainly for debugging purposes.

Flag: `Private Polymorphic Universes`

This flag, on by default, removes universes which appear only in the body of an opaque polymorphic definition from the definition's universe arguments. As such, no value needs to be provided for these universes when instantiating the definition. Universe constraints are automatically adjusted.

Consider the following definition:

```
Lemma foo@{i} : Type@{i}.
    1 subgoal

      ============================
      Type@{i}

Proof.
exact Type.
    No more subgoals.

Qed.
Print foo.
    foo@{i} =
    Type@{Top.63}
         : Type@{i}
    (* Public universes:
    i |= Set < i
    Private universes:
    {Top.63} |= Top.63 < i *)
```

The universe `Top.xxx` for the `Type` in the body cannot be accessed, we only care that one exists for any instantiation of the universes appearing in the type of `foo`. This is guaranteed when the transitive constraint `Set <= Top.xxx < i` is verified. Then when using the constant we don't need to put a value for the inner universe:

```
Check foo@{_}.
    foo@{Top.64}
         : Type@{Top.64}
    (* {Top.64} |= Set < Top.64 *)
```

and when not looking at the body we don't mention the private universe:

```
About foo.
    foo@{i} : Type@{i}
    (* i |= Set < i *)

    foo is universe polymorphic
    foo is opaque
    Expands to: Constant Top.foo
```

To recover the same behaviour with regard to universes as `Defined`, the *Private Polymorphic Universes* flag may be unset:

```
Unset Private Polymorphic Universes.
Lemma bar : Type.
    1 subgoal

      ============================
      Type@{Top.65}

Proof.
exact Type.
    No more subgoals.

Qed.
About bar.
    bar@{Top.65 Top.66} : Type@{Top.65}
    (* Top.65 Top.66 |= Top.66 < Top.65 *)

    bar is universe polymorphic
    bar is opaque
    Expands to: Constant Top.bar

Fail Check bar@{_}.
    The command has indeed failed with message:
    Universe instance should have length 2.

Check bar@{_ _}.
    bar@{Top.68
    Top.69}
         : Type@{Top.68}
    (* {Top.69 Top.68} |= Top.69 < Top.68 *)
```

Note that named universes are always public.

```
Set Private Polymorphic Universes.
Unset Strict Universe Declaration.
Lemma baz : Type@{outer}.
    1 subgoal

      ============================
      Type@{outer}
```

(continues on next page)

(continued from previous page)

```
Proof.
exact Type@{inner}.
    No more subgoals.

Qed.
About baz.
    baz@{outer inner} : Type@{outer}
    (* outer inner |= inner < outer *)

    baz is universe polymorphic
    baz is opaque
    Expands to: Constant Top.baz
```

8.14.9 Universe polymorphism and sections

Variables, *Context*, *Universe* and *Constraint* in a section support polymorphism. This means that the universe variables and their associated constraints are discharged polymorphically over definitions that use them. In other words, two definitions in the section sharing a common variable will both get parameterized by the universes produced by the variable declaration. This is in contrast to a “mononorphic” variable which introduces global universes and constraints, making the two definitions depend on the *same* global universes associated to the variable.

It is possible to mix universe polymorphism and monomorphism in sections, except in the following ways:

- no monomorphic constraint may refer to a polymorphic universe:

  ```
  Section Foo.
  Polymorphic Universe i.
  Fail Constraint i = i.
      The command has indeed failed with message:
      Cannot add monomorphic constraints which refer to section polymorphic universes.
  ```

  This includes constraints implicitly declared by commands such as *Variable*, which may need to be used with universe polymorphism activated (locally by attribute or globally by option):

  ```
  Fail Variable A : (Type@{i} : Type).
      The command has indeed failed with message:
      Cannot add monomorphic constraints which refer to section polymorphic universes.

  Polymorphic Variable A : (Type@{i} : Type).
      A is declared
  ```

  (in the above example the anonymous `Type` constrains polymorphic universe `i` to be strictly smaller.)

- no monomorphic constant or inductive may be declared if polymorphic universes or universe constraints are present.

These restrictions are required in order to produce a sensible result when closing the section (the requirement on constants and inductives is stricter than the one on constraints, because constants and inductives are abstracted by *all* the section’s polymorphic universes and constraints).

8.15 SProp (proof irrelevant propositions)

Warning: The status of strict propositions is experimental.

This section describes the extension of Coq with definitionally proof irrelevant propositions (types in the sort SProp, also known as strict propositions) as described in *[GCST19]*.

Using SProp may be prevented by passing `-disallow-sprop` to the Coq program or using *`Allow StrictProp`*.

Flag: `Allow StrictProp`

Allows using SProp when set and forbids it when unset. The initial value depends on whether you used the command line `-disallow-sprop` and `-allow-sprop`.

Error: `SProp not allowed, you need to Set Allow StrictProp or to use the -allow-sprop command-line-flag.`

Some of the definitions described in this document are available through `Coq.Logic.StrictProp`, which see.

8.15.1 Basic constructs

The purpose of SProp is to provide types where all elements are convertible:

```
Definition irrelevance (A:SProp) (P:A -> Prop) (x:A) (v:P x) (y:A) : P y := v.
```

Since we have definitional *η-expansion* for functions, the property of being a type of definitionally irrelevant values is impredicative, and so is SProp:

```
Check fun (A:Type) (B:A -> SProp) => (forall x:A, B x) : SProp.
```

Warning: Conversion checking through bytecode or native code compilation currently does not understand proof irrelevance.

In order to keep conversion tractable, cumulativity for SProp is forbidden:

```
Fail Check (fun (A:SProp) => A : Type).
    The command has indeed failed with message:
    In environment
    A : SProp
    The term "A" has type "SProp" while it is expected to have type "Type".
```

We can explicitly lift strict propositions into the relevant world by using a wrapping inductive type. The inductive stops definitional proof irrelevance from escaping.

```
Inductive Box (A:SProp) : Prop := box : A -> Box A.
Arguments box {_} _.
```

```
Fail Check fun (A:SProp) (x y : Box A) => eq_refl : x = y.
    The command has indeed failed with message:
    In environment
    A : SProp
    x : Box A
    y : Box A
```

(continues on next page)

(continued from previous page)

```
    The term "eq_refl" has type "x = x" while it is expected to have type
    "x = y" (cannot unify "x" and "y").

Definition box_irrelevant (A:SProp) (x y : Box A) : x = y
  := match x, y with box x, box y => eq_refl end.
```

In the other direction, we can use impredicativity to ”squash” a relevant type, making an irrelevant approximation.

```
Definition iSquash (A:Type) : SProp
  := forall P : SProp, (A -> P) -> P.
Definition isquash A : A -> iSquash A
  := fun a P f => f a.
Definition iSquash_sind A (P : iSquash A -> SProp) (H : forall x : A, P (isquash A x))
  : forall x : iSquash A, P x
  := fun x => x (P x) (H : A -> P x).
```

Or more conveniently (but equivalently)

```
Inductive Squash (A:Type) : SProp := squash : A -> Squash A.
```

Most inductives types defined in SProp are squashed types, i.e. they can only be eliminated to construct proofs of other strict propositions. Empty types are the only exception.

```
Inductive sEmpty : SProp := .

Check sEmpty_rect.
    sEmpty_rect
         : forall (P : sEmpty -> Type) (s : sEmpty), P s
```

Note: Eliminators to strict propositions are called `foo_sind`, in the same way that eliminators to propositions are called `foo_ind`.

Primitive records in SProp are allowed when fields are strict propositions, for instance:

```
Set Primitive Projections.
Record sProd (A B : SProp) : SProp := { sfst : A; ssnd : B }.
```

On the other hand, to avoid having definitionally irrelevant types in non-SProp sorts (through record η-extensionality), primitive records in relevant sorts must have at least one relevant field.

```
Set Warnings "+non-primitive-record".
Fail Record rBox (A:SProp) : Prop := rbox { runbox : A }.
    The command has indeed failed with message:
    The record rBox could not be defined as a primitive record

Record ssig (A:Type) (P:A -> SProp) : Type := { spr1 : A; spr2 : P spr1 }.
```

Note that `rBox` works as an emulated record, which is equivalent to the Box inductive.

8.15.2 Encodings for strict propositions

The elimination for unit types can be encoded by a trivial function thanks to proof irrelevance:

```
Inductive sUnit : SProp := stt.
Definition sUnit_rect (P:sUnit->Type) (v:P stt) (x:sUnit) : P x := v.
```

By using empty and unit types as base values, we can encode other strict propositions. For instance:

```
Definition is_true (b:bool) : SProp := if b then sUnit else sEmpty.

Definition is_true_eq_true b : is_true b -> true = b
  := match b with
     | true => fun _ => eq_refl
     | false => sEmpty_ind _
     end.

Definition eq_true_is_true b (H:true=b) : is_true b
  := match H in _ = x return is_true x with eq_refl => stt end.
```

8.15.3 Issues with non-cumulativity

During normal term elaboration, we don't always know that a type is a strict proposition early enough. For instance:

```
Definition constant_0 : ?[T] -> nat := fun _ : sUnit => 0.
```

While checking the type of the constant, we only know that `?[T]` must inhabit some sort. Putting it in some floating universe `u` would disallow instantiating it by `sUnit : SProp`.

In order to make the system usable without having to annotate every instance of SProp, we consider SProp to be a subtype of every universe during elaboration (i.e. outside the kernel). Then once we have a fully elaborated term it is sent to the kernel which will check that we didn't actually need cumulativity of SProp (in the example above, `u` doesn't appear in the final term).

This means that some errors will be delayed until `Qed`:

```
Lemma foo : Prop.
Proof.
pose (fun A : SProp => A : Type); exact True.


Fail Qed.
    The command has indeed failed with message:
    In environment
    A : SProp
    The term "A" has type "SProp" while it is expected to have type "Type".


Abort.
```

Flag: `Elaboration StrictProp Cumulativity`

Unset this flag (it is on by default) to be strict with regard to SProp cumulativity during elaboration.

The implementation of proof irrelevance uses inferred "relevance" marks on binders to determine which variables are irrelevant. Together with non-cumulativity this allows us to avoid retyping during conversion. However during elaboration cumulativity is allowed and so the algorithm may miss some irrelevance:

```
Fail Definition late_mark := fun (A:SProp) (P:A -> Prop) x y (v:P x) => v : P y.
    The command has indeed failed with message:
    In environment
```

(continues on next page)

(continued from previous page)

```
A : SProp
P : A -> Prop
x : A
y : A
v : P x
The term "v" has type "P x" while it is expected to have type "P y".
```

The binders for `x` and `y` are created before their type is known to be `A`, so they're not marked irrelevant. This can be avoided with sufficient annotation of binders (see `irrelevance` at the beginning of this chapter) or by bypassing the conversion check in tactics.

```
Definition late_mark := fun (A:SProp) (P:A -> Prop) x y (v:P x) =>
  ltac:(exact_no_check v) : P y.
```

The kernel will re-infer the marks on the fully elaborated term, and so correctly converts `x` and `y`.

Warning: `Bad relevance`

This is a developer warning, disabled by default. It is emitted by the kernel when it is passed a term with incorrect relevance marks. To avoid conversion issues as in `late_mark` you may wish to use it to find when your tactics are producing incorrect marks.

BIBLIOGRAPHY

[Asp00] David Aspinall. Proof general: a generic tool for proof development. In Susanne Graf and Michael Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000*, volume 1785 of Lecture Notes in Computer Science, pages 38–43. Springer Berlin Heidelberg, 2000. doi:10.1007/3-540-46419-0_3[319].

[Bar81] H.P. Barendregt. *The Lambda Calculus its Syntax and Semantics*. North-Holland, 1981.

[BDenesGregoire11] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of Lecture Notes in Computer Science, 362–377. Springer, 2011. URL: http://dx.doi.org/10.1007/978-3-642-25379-9_26, doi:10.1007/978-3-642-25379-9_26[320].

[Bou97] 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.

[Coq89] T. Coquand. Metamathematical investigations of a calculus of constructions. Technical Report RR-1088, INRIA, September 1989. URL: https://hal.inria.fr/inria-00075471.

[CH86a] T. Coquand and Gérard Huet. Concepts mathematiques et informatiques formalises dans le calcul des constructions. Technical Report RR-0515, INRIA, April 1986. URL: https://hal.inria.fr/inria-00076039.

[CH86b] T. Coquand and Gérard Huet. The calculus of constructions. Technical Report RR-0530, INRIA, May 1986. URL: https://hal.inria.fr/inria-00076024.

[Coq85] Th. Coquand. *Une Théorie des Constructions*. PhD thesis, Université Paris 7, January 1985.

[Coq86] Th. Coquand. An Analysis of Girard's Paradox. In *Symposium on Logic in Computer Science*. Cambridge, MA, 1986. IEEE Computer Society Press.

[Coq92] Th. Coquand. Pattern Matching with Dependent Types. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*. 1992.

[CH85] Thierry Coquand and Gérard Huet. Constructions: a higher order proof system for mechanizing mathematics. In *European Conference on Computer Algebra*, 151–184. Springer Berlin Heidelberg, 1985. URL: http://dx.doi.org/10.1007/3-540-15983-5_13, doi:10.1007/3-540-15983-5_13[321].

[319] https://doi.org/10.1007/3-540-46419-0_3

[320] https://doi.org/10.1007/978-3-642-25379-9_26

[321] https://doi.org/10.1007/3-540-15983-5_13

[CP90] Thierry Coquand and Christine Paulin. Inductively defined types. In *COLOG-88*, 50–66. Springer Berlin Heidelberg, 1990. URL: http://dx.doi.org/10.1007/3-540-52335-9_47, doi:10.1007/3-540-52335-9_47[322].

[CT95] Cristina Cornes and Delphine Terrasse. Automating inversion of inductive predicates in coq. In *TYPES*, 85–104. 1995.

[CFC58] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*. Volume 1. North-Holland, 1958. §9E.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, 207–212. New York, NY, USA, 1982. ACM. URL: http://doi.acm.org/10.1145/582153.582176, doi:10.1145/582153.582176[323].

[dB72] 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.*, 1972.

[Del00] 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, 85–95. Springer-Verlag, November 2000. URL: http://www.lirmm.fr/%7Edelahaye/papers/ltac%20(LPAR%2700).pdf.

[dC95] 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.

[Dyc92] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, September 1992.

[GCST19] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional Proof Irrelevance Without K. *Proc. ACM Program. Lang.*, 3(POPL):3:1–3:28, 2019. URL: http://doi.acm.org/10.1145/3290316.

[Gimenez94] 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.

[Gimenez95] 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, 135–152. Springer-Verlag, 1995.

[Gimenez98] E. Giménez. A tutorial on recursive types in coq. Technical Report, INRIA, March 1998.

[GimenezCasteran05] E. Giménez and P. Castéran. A tutorial on [co-]inductive types in coq. available at http://coq.inria.fr/doc, January 2005.

[GMN+91] Alessandro Giovini, Teo Mora, Gianfranco Niesi, Lorenzo Robbiano, and Carlo Traverso. ”one sugar cube, please” or selection strategies in the buchberger algorithm. In *Proceedings of the ISSAC'91, ACM Press*, 5–4. 1991.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989.

[GZND11] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. *SIGPLAN Not.*, 46(9):163–175, September 2011. URL: http://doi.acm.org/10.1145/2034574.2034798, doi:10.1145/2034574.2034798[324].

[322] https://doi.org/10.1007/3-540-52335-9_47

[323] https://doi.org/10.1145/582153.582176

[324] https://doi.org/10.1145/2034574.2034798

[GregoireL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, 235–246. ACM, 2002. URL: http://doi.acm.org/10.1145/581478.581501, doi:10.1145/581478.581501[325].

[How80] 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.

[Hue89] G. Huet. The Constructive Engine. In R. Narasimhan, editor, *A perspective in Theoretical Computer Science. Commemorative Volume for Gift Siromoney*. World Scientific Publishing, 1989.

[Hue88] Gérard Huet. Induction principles formalized in the calculus of constructions. In *Programming of Future Generation Computers. Elsevier Science*. Springer Berlin Heidelberg, 1988. URL: http://dx.doi.org/10.1007/3-540-17660-8_62, doi:10.1007/3-540-17660-8_62[326].

[LW11] Gyesik Lee and Benjamin Werner. Proof-irrelevant model of CC with predicative induction and judgmental equality. *Logical Methods in Computer Science*, 2011.

[Ler90] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[Let02] P. Letouzey. A new extraction for coq. In *TYPES*. 2002. URL: http://www.irif.fr/~letouzey/download/extraction2002.pdf.

[LV97] Sebastiaan P. Luttik and Eelco Visser. Specification of rewriting strategies. In *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97), Electronic Workshops in Computing*. Springer-Verlag, 1997.

[MT13] Assia Mahboubi and Enrico Tassi. Canonical Structures for the working Coq user. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of LNCS, 19–34. Rennes, France, 2013. Springer. URL: http://hal.inria.fr/hal-00816703, doi:10.1007/978-3-642-39634-2_5[327].

[McB00] Conor McBride. Elimination with a motive. In *TYPES*, 197–216. 2000.

[Moh86] Christine Mohring. Algorithm development in the calculus of constructions. In *LICS*, 84–91. 1986.

[Mun94] C. Muñoz. Démonstration automatique dans la logique propositionnelle intuitionniste. Master's thesis, DEA d'Informatique Fondamentale, Université Paris 7, September 1994.

[Mye86] Eugene Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1986. URL: http://www.xmailserver.org/diff2.pdf.

[Par95] 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.

[PM93a] 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.

[PM89] Christine Paulin-Mohring. Extracting ω's programs from proofs in the calculus of constructions. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 89–104. ACM Press, 1989. URL: http://dx.doi.org/10.1145/75277.75285, doi:10.1145/75277.75285[328].

[325] https://doi.org/10.1145/581478.581501
[326] https://doi.org/10.1007/3-540-17660-8_62
[327] https://doi.org/10.1007/978-3-642-39634-2_5
[328] https://doi.org/10.1145/75277.75285

[PM93b] Christine Paulin-Mohring. Inductive definitions in the system coq rules and properties. In *International Conference on Typed Lambda Calculi and Applications*, 328–345. Springer-Verlag, 1993. URL: http://dx.doi.org/10.1007/bfb0037116, doi:10.1007/bfb0037116[329].

[PPM89] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *International Conference on Mathematical Foundations of Programming Semantics*, 209–228. Springer-Verlag, 1989. URL: http://dx.doi.org/10.1007/bfb0040259, doi:10.1007/bfb0040259[330].

[PCC16] Clément Pit-Claudel and Pierre Courtieu. Company-coq: taking proof general one step closer to a real ide. In *CoqPL'16: The Second International Workshop on Coq for PL*. January 2016. doi:10.5281/zenodo.44331[331].

[Pug92] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communication of the ACM*, pages 102–114, 1992.

[ROS98] John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.

[Soz07] Matthieu Sozeau. Subset coercions in Coq. In *TYPES'06*, volume 4502 of LNCS, 237–252. Springer, 2007.

[SO08] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In *TPHOLs'08*. 2008.

[Vis01] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *RTA*, volume 2051 of LNCS, 357–362. 2001.

[VBT98] Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. Building program optimizers with rewriting strategies. In *ICFP*, 13–26. 1998.

[Wer94] B. Werner. *Une théorie des constructions inductives*. PhD thesis, Université Paris 7, 1994.

[329] https://doi.org/10.1007/bfb0037116

[330] https://doi.org/10.1007/bfb0040259

[331] https://doi.org/10.5281/zenodo.44331

COMMAND INDEX

TACTIC INDEX

FLAGS, OPTIONS AND TABLES INDEX

ERRORS AND WARNINGS INDEX

INDEX