The Coq Proof Assistant
Reference Manual

September 16, 2002

Version 7.3 [1]

The Coq Development Team

LogiCal Project

[1] This research was partly supported by ESPRIT Basic Research Action "Types"

V7.3, September 16, 2002

Introduction

This document is the Reference Manual of version V7.1 of the Coq proof assistant. A companion volume, the Coq Tutorial, is provided for the beginners. It is advised to read the Tutorial first.

The system Coq is designed to write formal specifications, programs and to verify that programs are correct with respect to their specification. It provides a specification language named Gallina. Terms of Gallina can represent programs as well as properties of these programs and proofs of these properties. Using the so-called *Curry-Howard isomorphism*, programs, properties and proofs are formalized 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. Other modes of interaction with Coq are possible, through an emacs shell window, or through a customized interface with the Centaur environment (CTCoq). These facilities are not documented here.

- The interactive mode may be used as a debugging mode in which the user can develop his theories and proofs step by step, backtracking if needed and so on. The interactive mode is run with the `coqtop` command from the operating system (which we shall assume to be some variety of UNIX in the rest of this document).

- The compiled mode acts as a proof checker taking a file containing a whole development in order to ensure its correctness. Moreover, Coq's compiler provides an output file containing a compact representation of its input. The compiled mode is run with the `coqc` command from the operating system. Its use is documented in chapter 11.

How to read this book

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

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

- The second part describes the proof engine. It is divided in three chapters. Chapter 5 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 the chapter 6. In chapter 7, all commands that realize one or more steps of the proof are presented: we call them *tactics*.
- The third part describes how to extend the system in two ways: adding parsing and pretty-printing rules (chapter 9) and writing new tactics (chapter 10)
- In the fourth part more practical tools are documented. First in the chapter 11 the usage of `coqc` (batch mode) and `coqtop` (interactive mode) with their options is described. Then (in chapter 12) various utilities that come with the Coq distribution are presented.

At the end of the document, after the global index, the user can find a tactic index and a vernacular command index.

List of additionnal documentation

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

Tutorial A companion volume to this reference manual, the Coq Tutorial, is aimed at gently introducing new users to developing proofs in Coq without assuming prior knowledge of type theory. In a second step, the user can read also the tutorial on recursive types (document `RecTutorial.ps`).

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

Installation A text file INSTALL that comes with the sources explains how to install Coq. A file UNINSTALL explains how uninstall or move it.

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

Credits

Coq is a proof assistant for higher-order logic, allowing the development of computer programs consistent with their formal specification. It is the result of about ten years of research of the Coq project. We shall briefly survey here three main aspects: the *logical language* in which we write our axiomatizations and specifications, the *proof assistant* which allows the development of verified mathematical proofs, and the *program extractor* which synthesizes computer programs obeying their formal specifications, written as logical assertions in the language.

The logical language used by Coq is a variety of type theory, called the *Calculus of Inductive Constructions*. Without going back to Leibniz and Boole, we can date the creation of what is now called mathematical logic to the work of Frege and Peano at the turn of the century. The discovery of antinomies in the free use of predicates or comprehension principles prompted Russell to restrict predicate calculus with a stratification of *types*. This effort culminated with *Principia Mathematica*, the first systematic attempt at a formal foundation of mathematics. A simplification of this system along the lines of simply typed λ-calculus occurred with Church's *Simple Theory of Types*. The λ-calculus notation, originally used for expressing functionality, could also be used as an encoding of natural deduction proofs. This Curry-Howard isomorphism was used by N. de Bruijn in the *Automath* project, the first full-scale attempt to develop and mechanically verify mathematical proofs. This effort culminated with Jutting's verification of Landau's *Grundlagen* in the 1970's. Exploiting this Curry-Howard isomorphism, notable achievements in proof theory saw the emergence of two type-theoretic frameworks; the first one, Martin-Löf's *Intuitionistic Theory of Types*, attempts a new foundation of mathematics on constructive principles. The second one, Girard's polymorphic λ-calculus F_ω, is a very strong functional system in which we may represent higher-order logic proof structures. Combining both systems in a higher-order extension of the Automath languages, T. Coquand presented in 1985 the first version of the *Calculus of Constructions*, CoC. This strong logical system allowed powerful axiomatizations, but direct inductive definitions were not possible, and inductive notions had to be defined indirectly through functional encodings, which introduced inefficiencies and awkwardness. The formalism was extended in 1989 by T. Coquand and C. Paulin with primitive inductive definitions, leading to the current *Calculus of Inductive Constructions*. This extended formalism is not rigorously defined here. Rather, numerous concrete examples are discussed. We refer the interested reader to relevant research papers for more information about the formalism, its meta-theoretic properties, and semantics. However, it should not be necessary to understand this theoretical material in order to write specifications. It is possible to understand the Calculus of Inductive Constructions at a higher level, as a mixture of predicate calculus, inductive predicate definitions presented as typed PROLOG, and recursive function definitions close to the language ML.

Automated theorem-proving was pioneered in the 1960's by Davis and Putnam in propositional calculus. A complete mechanization (in the sense of a semi-decision procedure) of classical first-order logic was proposed in 1965 by J.A. Robinson, with a single uniform inference rule called *resolution*. Resolution relies on solving equations in free algebras (i.e. term structures), using the *unification algorithm*. Many refinements of resolution were studied in the 1970's, but few convincing implementations were realized, except of course that PROLOG is in some sense issued from this effort. A less ambitious approach to proof development is computer-aided proof-checking. The most notable proof-checkers developed in the 1970's were LCF, designed by R. Milner and his colleagues at U. Edinburgh, specialized in proving properties about denotational semantics recursion equations, and the Boyer and Moore theorem-prover, an automation of primitive recursion over inductive data types. While the Boyer-Moore theorem-prover attempted to synthesize proofs by a combination of automated methods, LCF constructed its proofs through the programming of *tactics*, written in a high-level functional meta-language, ML.

The salient feature which clearly distinguishes our proof assistant from say LCF or Boyer and Moore's, is its possibility to extract programs from the constructive contents of proofs. This computational interpretation of proof objects, in the tradition of Bishop's constructive mathematics, is based on a realizability interpretation, in the sense of Kleene, due to C. Paulin. The user must just mark his intention by separating in the logical statements the assertions stating the existence of a computational object from the logical assertions which specify its properties, but which may be considered as just comments in the corresponding program. Given this information, the system automatically extracts a functional term from a consistency proof of its specifications. This functional term may be in turn compiled into an actual computer program. This methodology of extracting programs from proofs is a revolutionary paradigm for software engineering. Program synthesis has long been a theme of research in artificial intelligence, pioneered by R. Waldinger. The Tablog system of Z. Manna and R. Waldinger allows the deductive synthesis of functional programs from proofs in tableau form of their specifications, written in a variety of first-order logic. Development of a systematic *programming logic*, based on extensions of Martin-Löf's type theory, was undertaken at Cornell U. by the Nuprl team, headed by R. Constable. The first actual program extractor, PX, was designed and implemented around 1985 by S. Hayashi from Kyoto University. It allows the extraction of a LISP program from a proof in a logical system inspired by the logical formalisms of S. Feferman. Interest in this methodology is growing in the theoretical computer science community. We can foresee the day when actual computer systems used in applications will contain certified modules, automatically generated from a consistency proof of their formal specifications. We are however still far from being able to use this methodology in a smooth interaction with the standard tools from software engineering, i.e. compilers, linkers, run-time systems taking advantage of special hardware, debuggers, and the like. We hope that Coq can be of use to researchers interested in experimenting with this new methodology.

A first implementation of CoC was started in 1984 by G. Huet and T. Coquand. Its implementation language was CAML, a functional programming language from the ML family designed at INRIA in Rocquencourt. The core of this system was a proof-checker for CoC seen as a typed λ-calculus, called the *Constructive Engine*. This engine was operated through a high-level notation permitting the declaration of axioms and parameters, the definition of mathematical types and objects, and the explicit construction of proof objects encoded as λ-terms. A section mechanism, designed and implemented by G. Dowek, allowed hierarchical developments of mathematical theories. This high-level language was called the *Mathematical Vernacular*. Furthermore, an interactive *Theorem Prover* permitted the incremental construction of proof trees in a top-down manner, subgoaling recursively and backtracking from dead-alleys. The theorem prover executed tactics written in CAML, in the LCF fashion. A basic set of tactics was predefined, which the user could extend by his own specific tactics. This system (Version 4.10) was released in 1989. Then, the system was extended to deal with the new calculus with inductive types by C. Paulin, with corresponding new tactics for proofs by induction. A new standard set of tactics was streamlined, and the vernacular extended for tactics execution. A package to compile programs extracted from proofs to actual computer programs in CAML or some other functional language was designed and implemented by B. Werner. A new user-interface, relying on a CAML-X interface by D. de Rauglaudre, was designed and implemented by A. Felty. It allowed operation of the theorem-prover through the manipulation of windows, menus, mouse-sensitive buttons, and other widgets. This system (Version 5.6) was released in 1991.

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

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

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

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

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

Rocquencourt, Feb. 1st 1995
Gérard Huet

Credits: addendum for version 6.1

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

The new version provides powerful tools for easier developments.

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

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

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

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

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

Lyon, Nov. 18th 1996
Christine Paulin

Credits: addendum for version 6.2

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

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

Bruno Barras wrote new more efficient reductions functions.

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

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

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

Henri Laulhère produced the Coq distribution for the Windows environment.

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

Orsay, May 4th 1998
Christine Paulin

Credits: addendum for version 6.3

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

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

Orsay, Dec. 1999
Christine Paulin

Credits: versions 7

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

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 logical consistency check algorithm. He contributed to the simplification of Coq internal structures and the optimisation of the system. He added basic tactics for forward reasoning and coercions in patterns.

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

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

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

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

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

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

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

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

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

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

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

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. Chrząszcz, J. Courant, P. Courtieu, J. Duprat, J. Goubault, A. Miquel, C. Marché, B. Monate and B. Werner.

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

Orsay, May. 2002
Hugo Herbelin & Christine Paulin

Table of contents

Part I

The language

Chapter 1

The Gallina specification language

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

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

About the grammars in the manual

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

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

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

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

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

1.1 Lexical conventions

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

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

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

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

first_letter	::=	`a..z` | `A..Z` | _
subsequent_letter	::=	`a..z` | `A..Z` | `0..9` | _ | `'`
ident	::=	*first_letter* [*subsequent_letter*… *subsequent_letter*]

Identifiers can contain at most 80 characters, and all characters are meaningful. In particular, identifiers are case-sensitive. Access identifiers, written *access_ident*, are identifiers prefixed by `.` (dot). They are used in the syntax of qualified identifiers.

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

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

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

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

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

```
Add LoadPath "/usr/local/coq/\
contrib/Rocq/LAMBDA".
```

is correctly parsed, and equivalent to

```
Add LoadPath "/usr/local/coq/contrib/Rocq/LAMBDA".
```

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

```
as          end          in          of           using
with        Axiom        Cases       CoFixpoint   CoInductive
Compile     Definition   Fixpoint    Grammar      Hypothesis
Inductive   Load         Parameter   Proof        Prop
Qed         Quit         Set         Syntax       Theorem
Type        Variable
```

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

Add	AddPath	Abort	Abstraction	All
Begin	Cd	Chapter	Check	Compute
Defined	DelPath	Drop	End	Eval
Extraction	Fact	Focus	Goal	Guarded
Hint	Immediate	Induction	Infix	Inspect
Lemma	Let	LoadPath	Local	Minimality
ML	Module	Modules	Mutual	Opaque
Parameters	Print	Pwd	Remark	Remove
Require	Reset	Restart	Restore	Resume
Save	Scheme	Search	Section	Show
Silent	State	States	Suspend	Syntactic
Test	Transparent	Undo	Unset	Unfocus
Variables	Write			

Special tokens The following sequences of characters are special tokens:

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

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

1.2 Terms

1.2.1 Syntax of terms

Figure 1.1 describes the basic set of terms which form the *Calculus of Inductive Constructions* (also called CIC). The formal presentation of CIC is given in chapter 4. Extensions of this syntax are given in chapter 2. How to customize the syntax is described in chapter 9.

1.2.2 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 *identifiers*) are a syntactic subset of qualified identifiers. Identifiers may also denote local *variables*, what qualified identifiers do not.

1.2.3 Sorts

There are three sorts Set, Prop and Type.

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

term	::=	*qualid*
	|	*sort*
	|	*term* `->` *term*
	|	`(` *typed_idents* `;` ... `;` *typed_idents* `)` *term*
	|	`[` *local_decls* `;` ... `;` *local_decls* `]` *term*
	|	`(` *term* ... *term* `)`
	|	*[annotation]* `Cases` *term* `of` *[equation* | ... | *equation]* `end`
	|	`Fix` *ident* `{` *fix_body* `with` ... `with` *fix_body* `}`
	|	`CoFix` *ident* `{` *cofix_body* `with` ... `with` *cofix_body* `}`
qualid	::=	*ident*
	|	*qualid access_ident*
sort	::=	`Prop`
	|	`Set`
	|	`Type`
annotation	::=	`<` *term* `>`
typed_idents	::=	*ident* `,` ... `,` *ident* `:` *term*
local_assums	::=	*ident* `,` ... `,` *ident* *[*`:` *term]*
local_def	::=	*ident* `:=` *term* *[*`:` *term]*
local_decls	::=	*local_assums*
	|	*local_def*
fix_body	::=	*ident* `[` *typed_idents* `;` ... `;` *typed_idents* `]` `:` *term* `:=` *term*
cofix_body	::=	*ident* `:` *term* `:=` *term*
simple_pattern	::=	*ident*
	|	`(` *ident* ... *ident* `)`
equation	::=	*simple_pattern* `=>` *term*

Figure 1.1: Syntax of terms

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

- Type is the type of Set and Prop

More on sorts can be found in section 4.1.1.

1.2.4 Types

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

1.2.5 Abstractions

The expression "[*ident* : *type*] *term*" denotes the *abstraction* of the variable *ident* of type *type*, over the term *term*.

One can abstract several variables successively: the notation [$ident_1$, ... , $ident_n$: *type*] *term* stands for [$ident_1$: *type*](... ([$ident_n$: *type*] *term*) ...) and the notation [$local\_assums_1$; ... ; $local\_assums_m$] *term* is a shorthand for [$local\_assums_1$](... ([$local\_assums_m$] *term*).

Remark: The types of variables may be omitted in an abstraction when they can be synthesized by the system.

Remark: Local definitions may appear inside brackets mixed with assumptions. Obviously, this is expanded into unary abstractions separated by let-in's.

1.2.6 Products

The expression "(*ident* : *type*) *term*" denotes the *product* of the variable *ident* of type *type*, over the term *term*.

Similarly, the expression ($ident_1$, ... , $ident_n$: *type*) *term* is equivalent to ($ident_1$: *type*)(... (($ident_n$: *type*) *term*) ...) and the expression ($typed\_idents_1$; ... ; $typed\_idents_m$) *term* is a equivalent to ($typed\_idents_1$)(... (($typed\_idents_m$) *term*) ...)

1.2.7 Applications

($term_0$ $term_1$) denotes the application of term $term_0$ to $term_1$.

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

1.2.8 Local definitions (let-in)

[*ident*:=$term_1$] $term_2$ denotes the local binding of $term_1$ to the variable *ident* in $term_2$.

Remark: The expression [*ident*:=$term_1$:*type*]$term_2$ is an alternative form for the expression [*ident*:=($term_1$::*type*)]$term_2$.

Remark: An alternative equivalent syntax for let-in is `let` *ident* = *term* `in` *term*. For historical reasons, the syntax [*ident* = *term*] *term* is also available but is not recommended.

1.2.9 Definition by case analysis

In a simple pattern (*ident* ... *ident*), the first *ident* is intended to be a constructor.

The expression *[annotation]* `Cases` $term_0$ `of` $pattern_1$ => $term_1$ | ... | $pattern_n$ => $term_n$ `end`, denotes a *pattern-matching* over the term $term_0$ (expected to be of an inductive type).

The *annotation* is the resulting type of the whole `Cases` expression. Most of the time, when this type is the same as the types of all the $term_i$, the annotation is not needed[2]. The annotation has to be given when the resulting type of the whole `Cases` depends on the actual $term_0$ matched.

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

1.2.10 Recursive functions

The expression `Fix` *ident*$_i$ `{` *ident*$_1$ `[` *bindings*$_1$ `]` `:` *type*$_1$ `:=` *term*$_1$ `with` ... `with` *ident*$_n$ `[` *bindings*$_n$ `]` `:` *type*$_n$ `:=` *term*$_n$ `}` denotes the ith component of a block of functions defined by mutual well-founded recursion.

The expression `CoFix` *ident*$_i$ `{` *ident*$_1$ `:` *type*$_1$ `with` ... `with` *ident*$_n$ `[` *bindings*$_n$ `]` `:` *type*$_n$ `}` denotes the ith component of a block of terms defined by a mutual guarded recursion.

1.3 *The Vernacular*

Figure 1.3 describes *The Vernacular* which is the language of commands of Gallina. A sentence of the vernacular language, like in many natural languages, begins with a capital letter and ends with a dot. The different kinds of command are described hereafter. They all suppose that the terms occurring in the sentences are well-typed.

1.3.1 Declarations

The declaration mechanism allows the user to specify his own basic objects. Declared objects play the role of axioms or parameters in mathematics. A declared object is an *ident* associated to a *term*. A declaration is accepted by Coq if and only if this *term* is a correct type in the current context of the declaration and *ident* was not previously defined in the same module. This *term* is considered to be the type, or specification, of the *ident*.

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

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

Error messages:

1. `Clash with previous constant` *ident*

Variants:

1. `Parameter` *ident* `:` *term* `.`
   Is equivalent to `Axiom` *ident* `:` *term*
2. `Parameter` *ident*`,...,`*ident* `:` *term* `;` `...` `;` *ident*`,...,`*ident* `:` *term* `.`
   Links the *term*'s to the names comprising the lists *ident* `,` ... `,` *ident* : *term* `;` ... `;` *ident* `,` ... `,` *ident* : *term*.

Remark: It is possible to replace `Parameter` by `Parameters` when more than one parameter are given.

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

This command links *term* to the name *ident* in the context of the current section (see 2.4 for a description of the section mechanism). The name *ident* will be unknown when the current section will be closed. One says that the variable is *discharged*. Using the `Variable` command out of any section is equivalent to `Axiom`.

Error messages:

```
sentence              ::=  declaration
                        |  definition
                        |  statement
                        |  inductive
                        |  fixpoint
                        |  statement   proof

params                ::=  typed_idents ; ... ; typed_idents

declaration           ::=  Axiom ident : term .
                        |  declaration_keyword params .

declaration_keyword   ::=  Parameter | Parameters
                        |  Variable | Variables
                        |  Hypothesis | Hypotheses

definition            ::=  Definition ident [: term] := term .
                        |  Local ident [: term] := term .

inductive             ::=  [Mutual] Inductive ind_body with ... with ind_body .
                        |  [Mutual] CoInductive ind_body with ... with ind_body .

ind_body              ::=  ident [[ params ]] : term := [constructor | ... | constructor]

constructor           ::=  ident : term

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

statement             ::=  Theorem ident : term .
                        |  Lemma ident : term .
                        |  Definition ident : term .

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

Figure 1.2: Syntax of sentences

1. `Clash with previous constant` *ident*

Variants:

1. `Variable` *ident* `, ... ,` *ident*`:`*term* `; ... ;` *ident* `, ... ,` *ident*`:`*term* `.`
   Links *term* to the names comprising the list *ident* `, ... ,` *ident*:*term* `; ... ;` *ident* `, ... ,` *ident*:*term*

2. `Hypothesis` *ident* `, ... ,` *ident* `:` *term* `; ... ;` *ident* `, ... ,` *ident* `:` *term* `.`
   `Hypothesis` is a synonymous of `Variable`

Remark: It is possible to replace `Variable` by `Variables` and `Hypothesis` by `Hypotheses` when more than one variable or one hypothesis are given.

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

1.3.2 Definitions

Definitions differ from declarations since they allow to give a name to a term whereas declarations were just giving a type to a name. That is to say that the name of a defined object can be replaced at any time by its definition. This replacement is called δ-conversion (see section 4.3). A defined object is accepted by the system if and only if the defining term is well-typed in the current context of the definition. Then the type of the name is the type of term. The defined name is called a *constant* and one says that *the constant is added to the environment*.

A formal presentation of constants and environments is given in section 4.2.

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

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

Error messages:

1. `Clash with previous constant` *ident*

Variants:

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

Error messages:

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

See also: sections 5.2.4, 5.2.5, 7.5.5

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

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

Error messages:

1. `Clash with previous constant` *ident*

Variants:

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

See also: 2.4 (section mechanism), 5.2.4, 5.2.5 (opaque/transparent constants), 7.5.5

1.3.3 Inductive definitions

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

Simple inductive types

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

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

The name *ident* is the name of the inductively defined type and *sort* is the universes where it lives. The names $ident_1, \ldots, ident_n$ are the names of its constructors and $type_1, \ldots, type_n$ their respective types. The types of the constructors have to satisfy a *positivity condition* (see section 4.5.3) for *ident*. This condition ensures the soundness of the inductive definition. If this is the case, the constants *ident*, $ident_1, \ldots, ident_n$ are added to the environment with their respective types. Accordingly to the universe where the inductive type lives(*e.g.* its type *sort*), Coq provides a number of destructors for *ident*. Destructors are named *ident*`_ind`, *ident*`_rec` or *ident*`_rect` which respectively correspond to elimination principles on `Prop`, `Set` and `Type`. The type of the destructors expresses structural induction/recursion principles over objects of *ident*. We give below two examples of the use of the `Inductive` definitions.

The set of natural numbers is defined as:

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

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

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

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

This is the well known structural induction principle over natural numbers, i.e. the second-order form of Peano's induction principle. It allows to prove some universal property of natural numbers (`(n:nat)(P n)`) by induction on `n`. Recall that `(n:nat)(P n)` is Gallina's syntax for the universal quantification $\forall n : nat \cdot P(n)$.
The types of `nat_rec` and `nat_rect` are similar, except that they pertain to `(P:nat->Set)` and `(P:nat->Type)` respectively . They correspond to primitive induction principles (allowing dependent types) respectively over sorts `Set` and `Type`. 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).

Simple annotated inductive types

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

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

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

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

```
Coq < Check even_ind.
even_ind
     : (P:(nat->Prop))
        (P O)
        ->((n:nat)(even n)->(P n)->(P (S (S n))))
        ->(n:nat)(even n)->(P n)
```

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

Error messages:

1. `Non strictly positive occurrence of` *ident* `in` *type*

2. `Type of Constructor not well-formed`

Variants:

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

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

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

   ```
   Coq < Check Nil.
   Nil
        : (X:Set)(list X)
   ```

   and

   ```
   Coq < Check Cons.
   Cons
        : (X:Set)X->(list X)->(list X)
   ```

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

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

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

See also: sections 4.5, 2

Mutually inductive types

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

`Inductive` $ident_1$ `:` $type_1$ `:=`
    $ident^1_1$ `:` $type^1_1$
  `|` `...`
  `|` $ident^1_{n_1}$ `:` $type^1_{n_1}$
`with`
 `...`
`with` $ident_m$ `:` $type_m$ `:=`
    $ident^m_1$ `:` $type^m_1$
  `|` `...`
  `|` $ident^m_{n_m}$ `:` $type^m_{n_m}$ `.`

Remark: The word `Mutual` can be optionally inserted in front of `Inductive`.

It has the same semantics as the above `Inductive` definition for each $ident_1, \ldots, ident_m$. All names $ident_1, \ldots, ident_m$ and $ident^1_1, \ldots, ident^m_{n_m}$ are simultaneously added to the environment.

Then well-typing of constructors can be checked. Each one of the $ident_1, \ldots, ident_m$ can be used on its own.

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

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

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

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

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

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

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

```
Coq < Check tree_rec.
tree_rec
     : (P:(tree->Set))((a:A; f:forest)(P (node a f)))->(t:tree)(P t)

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

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

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

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

See also: 2.4

Co-inductive types

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

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

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

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

```
Coq < Definition hd  := [x:Stream]Cases x of (Seq a s) => a  end.
hd is defined
```

```
Coq < Definition tl  := [x:Stream]Cases x of (Seq a s) => s  end.
tl is defined
```

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

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

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

1.3.4 Definition of recursive functions

`Fixpoint` *ident* `[` $ident_1$ `:` $type_1$ `]` `:` $type_0$ `:=` $term_0$

This command allows to define inductive objects using a fixed point construction. The meaning of this declaration is to define *ident* a recursive function with one argument $ident_1$ of type $term_1$ such that (*ident* $ident_1$) has type $type_0$ and is equivalent to the expression $term_0$. The type of the *ident* is consequently ($ident_1$: $type_1$)$type_0$ and the value is equivalent to [$ident_1$: $type_1$]$term_0$. The argument $ident_1$ (of type $type_1$) is called the *recursive variable* of *ident*. Its type should be an inductive definition.

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

```
Coq < Fixpoint add [n:nat] : nat->nat
Coq <    := [m:nat]Cases n of O => m | (S p) => (S (add p m)) end.
add is recursively defined
```

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

Variants:

1. `Fixpoint` *ident* `[` *params* `] :` $type_0$ `:=` $term_0$ `.`
   It declares a list of identifiers with their type usable in the type $type_0$ and the definition body $term_0$ and the last identifier in *params* is the recursion variable.

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

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

```
Coq < Fixpoint wrongplus [n:nat] : nat->nat
Coq <    := [m:nat]Cases m of O => n | (S p) => (S (wrongplus n p)) end.
Error:
Recursive call applied to an illegal term
The recursive definition wrongplus :=
[n,m:nat]Cases m of
           O => n
         | (S p) => (S (wrongplus n p))
         end is not well-formed
```

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

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

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

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

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

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

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

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

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

```
Coq < Fixpoint tree_size [t:tree] : nat :=
Coq <     Cases t of (node a f) => (S (forest_size f)) end
Coq < with forest_size [f:forest] : nat :=
Coq <     Cases f of (leaf b) => (S O)
Coq <           | (cons t f') => (plus (tree_size t) (forest_size f'))
Coq <          end.
```

A generic command `Scheme` is useful to build automatically various mutual induction principles. It is described in section 7.14.

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

The `CoFixpoint` command introduces a method for constructing an infinite object of a coinductive type. For example, the stream containing all natural numbers can be introduced applying the following method to the number `O`:

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

Coq < Definition hd  := [x:Stream]Cases x of (Seq a s) => a  end.

Coq < Definition tl  := [x:Stream]Cases x of (Seq a s) => s  end.

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

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

```
Coq < CoFixpoint filter  : (nat->bool)->Stream->Stream :=
Coq <    [p:nat->bool]
Coq <      [s:Stream]
Coq <        if (p (hd s)) then (Seq (hd s) (filter p (tl s)))
Coq <        else (filter p (tl s)).
```

Notice that the definition contains an unguarded recursive call of `filter` on the `else` branch of the test.

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 expression. Isolate, it is considered as a canonical expression which is completely evaluated. We can test this using the command `Eval`, which computes the normal forms of a term:

```
Coq < Eval Compute in (from O).
     = (CoFix from{from :  nat->Stream := [n:nat](Seq n (from (S n)))}
         O)
     : Stream

Coq < Eval Compute in (hd (from O)).
     = O
     : nat

Coq < Eval Compute in (tl (from O)).
     = (CoFix from{from :  nat->Stream := [n:nat](Seq n (from (S n)))}
         (S O))
     : Stream
```

As in the `Fixpoint` command (cf. section 1.3.4), it is possible to introduce a block of mutually dependent methods. The general syntax for this case is:

`CoFixpoint` $ident_1$ `:`$type_1$ `:=` $term_1$
`with`
`...`
`with` $ident_m$ `:` $type_m$ `:=` $term_m$

1.3.5 Statement and proofs

A statement claims a goal of which the proof is then interactively done using tactics. More on the proof editing mode, statements and proofs can be found in chapter 6.

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

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

After a statement, Coq needs a proof.

Variants:

1. `Lemma` *ident* `:` *type* `.`
   It is a synonymous of `Theorem`

2. `Remark` *ident* `:` *type* `.`
   Same as `Theorem` except that if this statement is in one or more levels of sections then the name *ident* will be accessible only prefixed by the sections names when the sections (see 2.4 and 2.6) will be closed.

3. `Fact` *ident* `:` *type* `.`
   Same as `Remark` except that the innermost section name is dropped from the full name.

4. `Definition` *ident* `:` *type* `.`
   Allow to define a term of type *type* using the proof editing mode. It behaves as `Theorem` except the defined term will be transparent (see 5.2.5, 7.5.5).

`Proof . ...Qed .`

A proof starts by the keyword `Proof`. Then Coq enters the proof editing mode until the proof is completed. The proof editing mode essentially contains tactics that are described in chapter 7. Besides tactics, there are commands to manage the proof editing mode. They are described in chapter 6. When the proof is completed it should be validated and put in the environment using the keyword `Qed`.

Error message:

1. `Clash with previous constant` *ident*

Remarks:

1. Several statements can be simultaneously opened.

2. Not only other statements but any vernacular command can be given within the proof editing mode. In this case, the command is understood as if it would have been given before the statements still to be proved.

3. `Proof` is recommended but can currently be omitted. On the opposite, `Qed` (or `Defined`, see below) is mandatory to validate a proof.

4. Proofs ended by `Qed` are declared opaque (see 5.2.4) and cannot be unfolded by conversion tactics (see 7.5). To be able to unfold a proof, you should end the proof by `Defined` (see below).

Variants:

1. `Proof . ...Defined .`
   Same as `Proof . ...Qed .` but the proof is then declared transparent (see 5.2.5), which means it can be unfolded in conversion tactics (see 7.5).

2. `Proof . ...Save.`
   Same as `Proof . ...Qed .`

3. `Goal` *type*... `Save` *ident*
   Same as `Lemma` *ident* `:` *type*... `Save.` This is intended to be used in the interactive mode. Conversely to named lemmas, anonymous goals cannot be nested.

Chapter 2

Extensions of Gallina

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

2.1 Record types

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

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

the identifier *ident* is the name of the defined record and *sort* is its type. The identifier $ident_0$ is the name of its constructor. If $ident_0$ is omitted, the default name `Build_`*ident* is used. The identifiers $ident_1$, .., $ident_n$ are the names of fields and $term_1$, .., $term_n$ their respective types. Remark that the type of $ident_i$ may depend on the previous $ident_j$ (for $j < i$). Thus the order of the fields is important. Finally, *params* are the parameters of the record.

More generally, a record may have explicitly defined (a.k.a. manifest) fields. For instance, `Record` *ident* `[` *params* `]` **:** *sort* := `{` $ident_1$ **:** $type_1$ **;** $ident_2$ **:=** $term_2$ **;.** $ident_3$ **:** $type_3$ `}` in which case the correctness of $type_3$ may rely on the instance $term_2$ of $ident_2$ and $term_2$ in turn may depend on $ident_1$.

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

```
Coq < Record Rat : Set := mkRat {
Coq <  sign     : bool;
```

sentence	::=	*record*
record	::=	`Record` *ident* *[*`[` *params* `]`*]* **:** *sort* **:=** *[ident]* `{` *[field* **;** … **;** *field]* `}` **.**
field	::=	*ident* **:** *type*
	|	*ident* **:=** *term*
	|	*ident* **:** *type* **:=** *term*

Figure 2.1: Syntax for the definition of `Record`

```
Coq <  top      : nat;
Coq <  bottom   : nat;
Coq <  Rat_bottom_cond : ~O=bottom;
Coq <  Rat_irred_cond:(x,y,z:nat)(mult x y)=top/\(mult x z)=bottom->x=(S O)}.
Rat is defined
Rat_rect is defined
Rat_ind is defined
Rat_rec is defined
```

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

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

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

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

Let us define the rational $1/2$.

```
Coq < Require Arith.

Coq < Theorem one_two_irred: (x,y,z:nat)(mult x y)=(1)/\(mult x z)=(2)->x=(1).

...

Coq < Qed.

Coq < Definition half := (mkRat true (1) (2) (O_S (1)) one_two_irred).
half is defined

Coq < Check half.
half
     : Rat
```

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

```
Coq < Eval Compute in (top half).
     = (1)
     : nat

Coq < Eval Compute in (bottom half).
     = (2)
     : nat

Coq < Eval Compute in (Rat_bottom_cond half).
     = (O_S (1))
     : ~(0)=(bottom half)
```

Warnings:

1. `Warning:` $ident_i$ `cannot be defined.`
   It can happens that the definition of a projection is impossible. This message is followed by an explanation of this impossibility. There may be three reasons:

nested_pattern	:=	*ident*
	|	_
	|	`(` *ident nested_pattern* ... *nested_pattern* `)`
	|	`(` *nested_pattern* `as` *ident* `)`
	|	`(` *nested_pattern* `,` *nested_pattern* `)`
	|	`(` *nested_pattern* `)`
mult_pattern	:=	nested_pattern ... nested_pattern
ext_eqn	:=	*mult_pattern* => *term*
term	:=	*[annotation]* `Cases` *term* ... *term* `of` *[ext_eqn* | ... | *ext_eqn]* `end`

Figure 2.2: extended Cases syntax.

(a) The name $ident_i$ already exists in the environment (see section 1.3.1).

(b) The body of $ident_i$ uses a incorrect elimination for *ident* (see sections 1.3.4 and 4.5.4).

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

Error messages:

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

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

See also: Coercions and records in section 14.9 of the chapter devoted to coercions.

2.2 Variants and extensions of `Cases`

2.2.1 ML-style pattern-matching

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

The extension just acts as a macro that is expanded during parsing into a sequence of `Cases` on simple patterns. Especially, a construction defined using the extended `Cases` is printed under its expanded form.

The syntax of the extended `Cases` is presented in figure 2.2. Note the annotation is mandatory when the sequence of equation is empty.

See also: chapter 13.

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

For inductive types isomorphic to the boolean types (i.e. two constructors without arguments), it is possible to use a `if ... then ... else` notation. This enriches the syntax of terms as follows:

term := *[annotation]* `if` *term* `then` *term* `else` *term*

For instance, the definition

```
Coq < Definition not := [b:bool] Cases b of true => false | false => true end.
not is defined
```

can be alternatively written

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

2.2.3 Irrefutable patterns: the destructuring `let`

Terms in an inductive type having only one constructor, say `foo`, have necessarily the form `(foo ...)`. In this case, the `Cases` construction can be replaced by a `let ... in ...` construction. This enriches the syntax of terms as follows:

| *[annotation]* `let ( ` *ident* `,` ... `,` *ident* ` )` = *term* `in` *term*

For instance, the definition

```
Coq < Definition fst := [A,B:Set][H:A*B] Cases H of (pair x y) => x end.
fst is defined
```

can be alternatively written

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

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

2.2.4 Options for pretty-printing of `Cases`

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

Printing of wildcard pattern

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

```
Set Printing Wildcard.
```

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

`Unset Printing Wildcard.`

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

`Test Printing Wildcard.`

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

Printing of the elimination predicate

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

`Set Printing Synth.`

The result type is not printed when it is easily synthesizable.

`Unset Printing Synth.`

This forces the result type to be always printed (and then between angle brackets).

`Test Printing Synth.`

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

Printing matching on irrefutable pattern

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

`Add Printing Let` *ident*`.`

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

`Remove Printing Let` *ident*`.`

This removes *ident* from this list.

`Test Printing Let` *ident*`.`

This tells if *ident* belongs to the list.

`Print Table Printing Let.`

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

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

Printing matching on booleans

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

`Add Printing If` *ident*`.`

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

`Remove Printing If` *ident*`.`

This removes *ident* from this list.

`Test Printing If` *ident*`.`

This tells if *ident* belongs to the list.

`Print Table Printing If.`

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

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

Example

This example emphasizes what the printing options offer.

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

Coq < Print fst.
fst = [A,B:Set; p:(A*B)](let (x, _) = p in x)
     : (A,B:Set)A*B->A

Coq < Remove Printing Let prod.

Coq < Unset Printing Synth.

Coq < Unset Printing Wildcard.

Coq < Print snd.
snd =
[A,B:Set; p:(A*B)]<[_:(A*B)]B>Cases p of (pair x y) => y end
     : (A,B:Set)A*B->B
```

2.3 Forced type

In some cases, one want to assign a particular type to a term. The syntax to force the type of a term is the following:

term ::= (*term* : : *term*)

It forces the first term to be of type the second term. The type must be compatible with the term. More precisely it must be either a type convertible to the automatically inferred type (see chapter 4) or a type coercible to it, (see 2.8). When the type of a whole expression is forced, it is usually not necessary to give the types of the variables involved in the term.

Example:

```
Coq < Definition ID := (X:Set) X -> X.
ID is defined

Coq < Definition id := (([X][x]x) :: ID).
id is defined

Coq < Check id.
id
     : ID
```

2.4 Section mechanism

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

2.4.1 `Section` *ident*

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

Variants:

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

2.4.2 `End` *ident*

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

Here is an example :

```
Coq < Section s1.

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

Coq < Local y' := y.
y' is defined
```

```
Coq < Definition x' := (S x).
x' is defined

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

Coq < End s1.
x' is discharged.

Coq < Print x'.
x' = [x:nat](S x)
     : nat->nat
```

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

Error messages:

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

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

Remarks:

1. Most commands, like `Hint` *ident* or `Syntactic Definition` which appear inside a section are cancelled when the section is closed.

2.5 Logical paths of libraries and compilation units

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

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

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

Physical paths vs logical paths Libraries and sub-libraries are denoted by *logical directory paths* (written *dirpath* and of which the syntax is the same as *qualid*, see 1.2.2). Logical directory paths can be mapped to physical directories of the operating system using the command (see 5.5.3)

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

A library can inherit the tree structure of a physical directory by using the `-R` option to `coqtop` or the command (see 5.5.4)

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

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

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

2.6 Qualified names

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

Absolute and short names The full name of a library, module, section, definition, theorem, ... is called *absolute name*. The final identifier (in the example above, it is `eq`) is called *short name* (or sometimes *base name*). Coq maintains a *names table* mapping short names to absolute names. This greatly simplifies the notations. Coq cannot accept two constructions (definition, theorem, ...) with the same absolute name.

The special case of remarks and facts In contrast with definitions, lemmas, theorems, axioms and parameters, the absolute name of remarks includes the segment of sections in which it is defined. Concretely, if a remark `R` is defined in subsection `S2` of section `S1` in module `M`, then its absolute name is `M.S1.S2.R`. The same for facts, except that the name of the innermost section is dropped from the full name. Then, if a fact `F` is defined in subsection `S2` of section `S1` in module `M`, then its absolute name is `M.S1.F`.

Visibility and qualified names An absolute name is called *visible* when its base name suffices to denote it. This means the base name is mapped to the absolute name in Coq name table.

All the base names of definitions and theorems are automatically put in the Coq name table. But sometimes, the short name of a construction defined in a module may hide the short name of another construction defined in another module. Instead of just distinguishing the clashing names by using the absolute names, it is enough to prefix the base name just by the name of the module in which the definition, theorem, ... is defined. Such a name built from single identifiers separated by dots is called a *partially qualified name* (or shortly a *qualified name*, written *qualid*). Especially, both absolute names and short names are qualified names. To ensure that a construction always remains accessible, absolute names can never be hidden.

Examples:

```
Coq < Check O.
O
     : nat

Coq < Definition nat := bool.
nat is defined
```

```
Coq < Check O.
O
     : Datatypes.nat

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

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

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

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

2.7 Implicit arguments

The Coq system allows to skip during a function application certain arguments that can be automatically inferred from the other arguments. Such arguments are called *implicit*. Typical implicit arguments are the type arguments in polymorphic functions.

The user can force a subterm to be guessed by replacing it by `?`. If possible, the correct subterm will be automatically generated.

Error message:

1. `There is an unknown subterm I cannot solve`
   Coq was not able to deduce an instantiation of a "?".

In addition, there are two ways to systematically avoid to write "?" where a term can be automatically inferred.

The first mode is automatic. Switching to this mode forces some easy-to-infer subterms to always be implicit. The command to use the second mode is `Syntactic Definition`.

2.7.1 Auto-detection of implicit arguments

There is an automatic mode to declare as implicit some arguments of constants and variables which have a functional type. In this mode, to every declared object (even inductive types and theirs constructors) is associated the list of the positions of its implicit arguments. These implicit arguments correspond to the arguments which can be deduced from the following ones. Thus when one applies these functions to arguments, one can omit the implicit ones. They are then automatically replaced by symbols "?", to be inferred by the mechanism of synthesis of implicit arguments.

`Set Implicit Arguments.`

This command switches the automatic implicit arguments mode on. To switch it off, use `Unset Implicit Arguments.`. The mode is off by default.

The computation of implicit arguments takes account of the unfolding of constants. For instance, the variable `p` below has a type `(Transitivity R)` which is reducible to `(x,y:U)(R x y) -> (z:U)(R y z) -> (R x z)`. As the variables `x`, `y` and `z` appear in the body of the type, they are said implicit; they correspond respectively to the positions `1`, `2` and `4`.

```
Coq < Set Implicit Arguments.

Coq < Variable X : Type.

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

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

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

Coq < Print p.
*** [ p : (Transitivity R) ]
Positions [1; 2; 4] are implicit

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

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

Explicit Applications

The mechanism of synthesis of implicit arguments is not complete, so we have sometimes to give explicitly certain implicit arguments of an application. The syntax is i`!`*term* where i is the position of an implicit argument and *term* is its corresponding explicit term. The number i is called *explicitation number*. We can also give all the arguments of an application, we have then to write `(!`*ident* *term*$_1$`..`*term*$_n$`)`.

Error message:

1. `Bad explicitation number`

Example:

```
Coq < Check (p r1 4!c).
(p r1 4!c)
     : (R b c)->(R a c)

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

Implicit Arguments and Pretty-Printing

The basic pretty-printing rules hide the implicit arguments of an application. However an implicit argument *term* of an application which is not followed by any explicit argument is printed as follows i!*term* where i is its position.

2.7.2 User-defined implicit arguments: `Syntactic definition`

The syntactic definitions define syntactic constants, i.e. give a name to a term possibly untyped but syntactically correct. Their syntax is:

`Syntactic Definition` *name* `:=` *term* `.`

Syntactic definitions behave like macros: every occurrence of a syntactic constant in an expression is immediately replaced by its body.

Let us extend our functional language with the definition of the identity function:

```
Coq < Definition explicit_id := [A:Set][a:A]a.
explicit_id is defined
```

We declare also a syntactic definition `id`:

```
Coq < Syntactic Definition id := (explicit_id ?).
id is now a syntax macro
```

The term `(explicit_id ?)` is untyped since the implicit arguments cannot be synthesized. There is no type check during this definition. Let us see what happens when we use a syntactic constant in an expression like in the following example.

```
Coq < Check (id O).
(explicit_id nat O)
     : nat
```

First the syntactic constant `id` is replaced by its body `(explicit_id ?)` in the expression. Then the resulting expression is evaluated by the typechecker, which fills in "`?`" place-holders.

The standard usage of syntactic definitions is to give names to terms applied to implicit arguments "`?`". In this case, a special command is provided:

`Syntactic Definition` *name* `:=` *term* `|` n `.`

The body of the syntactic constant is *term* applied to n place-holders "`?`".

We can define a new syntactic definition `id1` for `explicit_id` using this command. We changed the name of the syntactic constant in order to avoid a name conflict with `id`.

```
Coq < Syntactic Definition id1 := explicit_id | 1.
id1 is now a syntax macro
```

The new syntactic constant `id1` has the same behavior as `id`:

```
Coq < Check (id1 O).
(explicit_id nat O)
     : nat
```

Warnings:

1. Syntactic constants defined inside a section are no longer available after closing the section.
2. You cannot see the body of a syntactic constant with a `Print` command.

2.7.3 Canonical structures

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

`Canonical Structure` *qualid* `.`

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

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

```
Coq < Require Relations.

Coq < Require EqNat.

Coq < Set Implicit Arguments.

Coq < Structure Setoid : Type :=
Coq <  {Carrier   :> Set;
Coq <   Equal     : (relation Carrier);
Coq <   Prf_equiv : (equivalence Carrier Equal)}.

Coq < Definition is_law := [A,B:Setoid][f:A->B]
Coq <    (x,y:A) (Equal x y) -> (Equal (f x) (f y)).

Coq < Axiom eq_nat_equiv : (equivalence nat eq_nat).

Coq < Definition nat_setoid : Setoid := (Build_Setoid eq_nat_equiv).

Coq < Canonical Structure nat_setoid.
```

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

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

  ============================
   (is_law S)
```

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

Variants:

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

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

   `Canonical Structure` *ident*.

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

2.8 Implicit 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 $(x_1 : A_1)..(x_n : A_n)s$ 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 commands related to coercions are provided in chapter 14.

Chapter 3

The Coq library

The Coq library is structured into three parts:

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

The standard library: general-purpose libraries containing various developments of Coq axiomatizations about sets, lists, sorting, arithmetic, etc. This library comes with the system and its modules are directly accessible through the `Require` command (see section 5.4.2);

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

This chapter briefly reviews these libraries.

3.1 The basic library

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

3.1.1 Logic

The basic library of Coq comes with the definitions of standard (intuitionistic) logical connectives (they are defined as inductive constructions). They are equipped with an appealing syntax enriching the (subclass *form*) of the syntactic class *term*. The syntax extension [2] is shown on figure 3.1.1.

Propositional Connectives

First, we find propositional calculus connectives:

[1] These constructions are defined in the `Prelude` module in directory `theories/Init` at the Coq root directory; this includes the modules `Logic`, `Datatypes`, `Specif`, `Peano`, and `Wf` plus the module `Logic_Type`

[2] This syntax is defined in module `LogicSyntax`

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

Remark: The 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. There is true reason to have the notation `(` `ALL` *ident [*`:` *specif]* | *form* `)` propositions), except to have a notation dual of the notation for first-order existential quantification.

Figure 3.1: Syntax of formulas

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

Coq < Inductive False : Prop := .

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

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

Coq < Section Projections.

Coq < Variables A,B : Prop.

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

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


Coq < End Projections.

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

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

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

Quantifiers

Then we find first-order quantifiers:

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

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

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

The following abbreviations are allowed:

`(ALL x:A | P)`	`(all A [x:A]P)`
`(ALL x | P)`	`(all A [x:A]P)`
`(EX x:A | P)`	`(ex A [x:A]P)`
`(EX x | P)`	`(ex A [x:A]P)`
`(EX x:A | P & Q)`	`(ex2 A [x:A]P [x:A]Q)`
`(EX x | P & Q)`	`(ex2 A [x:A]P [x:A]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 `Set A` and an `x` of type `A`, the predicate `(eq A x)` is the smallest one which contains `x`. This definition, due to Christine Paulin-Mohring, is equivalent to define `eq` as the smallest reflexive relation, and it is also equivalent to Leibniz' equality.

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

Lemmas

Finally, a few easy lemmas are provided.

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


Coq < Section equality.
Coq <  Variable A,B : Set.
Coq <  Variable f   : A->B.
Coq <  Variable x,y,z : A.
Coq <  Theorem sym_eq : x=y -> y=x.
Coq <  Theorem trans_eq : x=y -> y=z -> x=z.
Coq <  Theorem f_equal : x=y -> (f x)=(f y).
Coq <  Theorem sym_not_eq : ~(x=y) -> ~(y=x).


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

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

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

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

3.1.2 Datatypes

In the basic library, we find the definition[3] of the basic data-types of programming, again defined as inductive constructions over the sort `Set`. Some of them come with a special syntax shown on figure 3.1.3.

Programming

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

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

Coq < Inductive option [A:Set] : Set := Some : A -> (option A)
Coq <                                 | None : (option A).

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

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

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

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

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

Coq < Section projections.

Coq <    Variables A,B:Set.

Coq <    Definition fst := [H:(prod A B)] Cases H of (pair x y) => x end.

Coq <    Definition snd := [H:(prod A B)] Cases H of (pair x y) => y end.

Coq < End projections.

Coq < Syntactic Definition Fst := (fst ? ?).

Coq < Syntactic Definition Snd := (snd ? ?).
```

[3] They are in `Datatypes.v`

3.1.3 Specification

The following notions[4] allows to build new datatypes and specifications. They are available with the syntax shown on figure 3.1.3[5].

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

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

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

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

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

```
Coq < Inductive sigS [A:Set;P:A->Set] : Set
Coq <      := existS : (x:A)(P x) -> (sigS A P).

Coq < Section sigSprojections.

Coq <    Variable A:Set.

Coq <    Variable P:A->Set.

Coq <    Definition projS1 := [H:(sigS A P)] let (x,h) = H in x.

Coq <    Definition projS2 := [H:(sigS A P)]<[H:(sigS A P)](P (projS1 H))>
Coq <                              let (x,h) = H in h.

Coq < End sigSprojections.

Coq < Inductive sigS2 [A:Set;P,Q:A->Set] : Set
Coq <      := existS2 : (x:A)(P x) -> (Q x) -> (sigS2 A P Q).
```

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

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

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

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

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* }	(`sigS`)
	|	{ *ident* : *specif* & *specif* & *specif* }	(`sigS2`)
term	::=	(*term* , *term*)	(`pair`)

Figure 3.2: Syntax of datatypes and specifications

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

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

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

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

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

```
Coq < Definition Exc := option.
```

```
Coq < Definition value := Some.
```

```
Coq < Definition error := None.
```

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

```
Coq < Definition except := False_rec.
```

```
Coq < Syntactic Definition Except := (except ?).
```

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

```
Coq < Theorem and_rec : (A,B:Prop)(C:Set)(A->B->C)->(A/\B)->C.
```

3.1.4 Basic Arithmetics

The basic library includes a few elementary properties of natural numbers, together with the definitions of predecessor, addition and multiplication[6].

[4] They are defined in module `Specif.v`
[5] This syntax can be found in the module `SpecifSyntax.v`
[6] This is in module `Peano.v`

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

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

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

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

Coq < Hints Immediate eq_add_S : core.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.1.5 Well-founded recursion

The basic library contains the basics of well-founded recursion and well-founded induction[7].

```
Coq < Chapter Well_founded.

Coq < Variable A : Set.

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

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

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

Coq < Section AccRec.

Coq < Variable P : A -> Set.

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

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

Coq < End AccRec.

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

Coq < Hypothesis Rwf : well_founded.

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

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

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

```
Coq < Section FixPoint.

Coq < Variable P : A -> Set.

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

Coq < Fixpoint Fix_F [x:A;r:(Acc x)] : (P x) :=
Coq <           (F x [y:A][p:(R y x)](Fix_F y (Acc_inv x r y p))).

Coq < Definition fix := [x:A](Fix_F x (Rwf x)).

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

Coq < Lemma Fix_F_eq
Coq <   : (x:A)(r:(Acc x))
Coq <     (F x [y:A][p:(R y x)](Fix_F y (Acc_inv x r y p)))=(Fix_F x r).

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

Coq < Lemma fix_eq : (x:A)(fix x)=(F x [y:A][p:(R y x)](fix y)).

Coq < End FixPoint.

Coq < End Well_founded.
```

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

3.1.6 Accessing the Type level

The basic library includes the definitions[8] of logical quantifiers axiomatized at the `Type` level.

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

Coq < Section universal_quantification.

Coq < Variable A : Type.

Coq < Variable P : A->Prop.

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

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

Coq < End universal_quantification.

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

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

It defines also Leibniz equality `x==y` when `x` and `y` belong to `A:Type`.

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

Coq < Section Equality_is_a_congruence.

Coq < Variables A,B : Type.

Coq < Variable  f : A->B.

Coq < Variable x,y,z : A.

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

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

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

Coq < End Equality_is_a_congruence.

Coq < Hints Immediate sym_eqT sym_not_eqT : core.

Coq < Definition eqT_ind_r: (A:Type)(x:A)(P:A->Prop)(P x)->(y:A)y==x -> (P y).
```

The figure 3.1.6 presents the syntactic notations corresponding to the main definitions [9]
At the end, it defines datatypes at the Type level.

```
Coq < Inductive EmptyT: Type :=.

Coq < Inductive UnitT : Type := IT : UnitT.

Coq < Definition notT := [A:Type] A->EmptyT.

Coq <
Coq < Inductive identityT [A:Type; a:A] : A->Type :=
Coq <      refl_identityT : (identityT A a a).
```

[8]This is in module `Logic_Type.v`
[9]This syntax is defined in module `Logic_TypeSyntax`

form	::=	(`ALLT` *ident* [`:` *specif*] | *form*)	(`allT`)
	|	(`EXT` *ident* [`:` *specif*] | *form*)	(`exT`)
	|	(`EXT` *ident* [`:` *specif*] | *form* `&` *form*)	(`exT2`)
	|	*term* == *term*	(`eqT`)

Figure 3.3: Syntax of first-order formulas in the type universe

3.2 The standard library

3.2.1 Survey

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

Logic	Classical logic and dependent equality
Arith	Basic Peano arithmetic
ZArith	Basic integer arithmetic
Bool	Booleans (basic functions and results)
Lists	Monomorphic and polymorphic lists (basic functions and results), Streams (infinite sequences defined with co-inductive types)
Sets	Sets (classical, constructive, finite, infinite, power set, etc.)
IntMap	Representation of finite sets by an efficient structure of map (trees indexed by binary integers).
Reals	Axiomatization of Real Numbers (classical, basic functions, integer part, fractional part, limit, derivative, Cauchy series, power series and results,... Requires the **ZArith** library).
Relations	Relations (definitions and basic results).
Sorting	Sorted list (basic definitions and heapsort correctness).
Wellfounded	Well-founded relations (basic results).

These directories belong to the initial load path of the system, and the modules they provide are compiled at installation time. So they are directly accessible with the command `Require` (see chapter 5).

The different modules of the Coq standard library are described in the additional document `Library.dvi`. They are also accessible on the WWW through the Coq homepage [10].

3.2.2 Notations for integer arithmetics

On figure 3.2.2 is described the syntax of expressions for integer arithmetics. It is provided by requiring the module `ZArith`.

The + and − binary operators bind less than the `*` operator which binds less than the | ... | and − unary operators which bind less than the others constructions. All the binary operators are left associative. The `[` ... `]` allows to escape the *zarith* grammar.

3.2.3 Notations for Peano's arithmetic (`nat`)

After having required the module `Arith`, the user can type the naturals using decimal notation. That is he can write `(3)` for `(S (S (S O)))`. The number must be between parentheses. This

[10] `http://coq.inria.fr`

<table>
<tr><td>form</td><td>::=</td><td>` <i>zarith_formula</i> `</td></tr>
<tr><td>term</td><td>::=</td><td>` <i>zarith</i> `</td></tr>
<tr><td>zarith_formula</td><td>::=</td><td>zarith = zarith</td></tr>
<tr><td></td><td>|</td><td>zarith <= zarith</td></tr>
<tr><td></td><td>|</td><td>zarith < zarith</td></tr>
<tr><td></td><td>|</td><td>zarith >= zarith</td></tr>
<tr><td></td><td>|</td><td>zarith > zarith</td></tr>
<tr><td></td><td>|</td><td>zarith = zarith = zarith</td></tr>
<tr><td></td><td>|</td><td>zarith <= zarith <= zarith</td></tr>
<tr><td></td><td>|</td><td>zarith <= zarith < zarith</td></tr>
<tr><td></td><td>|</td><td>zarith < zarith <= zarith</td></tr>
<tr><td></td><td>|</td><td>zarith < zarith < zarith</td></tr>
<tr><td></td><td>|</td><td>zarith <> zarith</td></tr>
<tr><td></td><td>|</td><td>zarith ? = zarith</td></tr>
<tr><td>zarith</td><td>::=</td><td>zarith + zarith</td></tr>
<tr><td></td><td>|</td><td>zarith - zarith</td></tr>
<tr><td></td><td>|</td><td>zarith * zarith</td></tr>
<tr><td></td><td>|</td><td>| zarith |</td></tr>
<tr><td></td><td>|</td><td>- zarith</td></tr>
<tr><td></td><td>|</td><td>ident</td></tr>
<tr><td></td><td>|</td><td>[term]</td></tr>
<tr><td></td><td>|</td><td>(zarith ... zarith)</td></tr>
<tr><td></td><td>|</td><td>(zarith , ... , zarith)</td></tr>
<tr><td></td><td>|</td><td>integer</td></tr>
</table>

Figure 3.4: Syntax of expressions in integer arithmetics

works also in the left hand side of a `Cases` expression (see for example section 8.1).

```
Coq < Require Arith.

Coq < Fixpoint even [n:nat] : bool :=
Coq < Cases n of (0) => true
Coq <          |  (1) => false
Coq <          | (S (S n)) => (even n)
Coq < end.
```

3.2.4 Real numbers library

Notations for real numbers

This is provided by requiring the module `Reals`. This notation is very similar to the notation for integer arithmetics (see figure 3.2.2) where Inverse (/x) and division (x/y) have been added. This syntax is used parenthesizing by a double back-quote (``).

```
Coq < Require Reals.

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

A constant, say ``` ``4`` ```, is equivalent to ``` ``(1+(1+(1+1)))`` ```.

Some tactics

In addition to the Ring, Field and Fourier tactics (see chapter 7) there are:

`DiscrR` prove that a real integer constante c1 is non equal to another real integer constante c2.

```
Coq < Require DiscrR.

Coq < Goal ``5<>0``.

Coq < DiscrR.
Subtree proved!
```

`SplitAbsolu` allows us to unfold `Rabsolu` contants and split corresponding conjonctions.

```
Coq < Require SplitAbsolu.

Coq < Goal (x:R) ``x <= (Rabsolu x)``.

Coq < Intro;SplitAbsolu.
2 subgoals

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

SplitRmult allows us to split a condition that a product is non equal to zero into subgoals corresponding to the condition on each subterm of the product.

```
Coq < Require SplitRmult.

Coq < Goal (x,y,z:R)``x*y*z<>0``.

Coq < Intros;SplitRmult.
3 subgoals

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

All this tactics has been written with the new tactic language.

3.3 Users' contributions

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

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

    http://coq.inria.fr/contribs-eng.html

Chapter 4

The Calculus of Inductive Constructions

The underlying formal language of Coq is the *Calculus of (Co)Inductive Constructions* (CIC in short). It is presented in this chapter.

In CIC all objects have a *type*. There are types for functions (or programs), there are atomic types (especially datatypes)... but also types for proofs and types for the types themselves. Especially, any object handled in the formalism must belong to a type. For instance, the statement *"for all x, P"* is not allowed in type theory; you must say instead: *"for all x belonging to T, P"*. The expression *"x belonging to T"* is written *"x:T"*. One also says: *"x has type T"*. The terms of CIC are detailed in section 4.1.

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

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

The reader seeking a background on the CIC may read several papers. Giménez [56] provides an introduction to inductive and coinductive definitions in Coq, Werner [107] and Paulin-Mohring [96] are the most recent theses on the CIC. Coquand-Huet [22, 23, 24] introduces the Calculus of Constructions. Coquand-Paulin [25] introduces inductive definitions. The CIC is a formulation of type theory including the possibility of inductive constructions. Barendregt [5] studies the modern form of type theory.

4.1 The terms

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

Consider for instance the $\rightarrow$ constructor and assume nat is the type of natural numbers. Then $\rightarrow$ is used both to denote $\mathsf{nat} \rightarrow \mathsf{nat}$ which is the type of functions from nat to nat, and to denote $\mathsf{nat} \rightarrow \mathsf{Prop}$ which is the type of unary predicates over the natural numbers. Consider abstraction which builds functions. It serves to build "ordinary" functions as $[x:\mathsf{nat}](\mathtt{mult}\ x\ x)$ (assuming `mult` is already defined) but may build also predicates over the natural numbers. For instance $[x:\mathsf{nat}](x=x)$ will represent a predicate P, informally written in mathematics $P(x) \equiv x = x$. If P has type $\mathsf{nat} \rightarrow \mathsf{Prop}$, $(P\ x)$ is a proposition, furthermore $(x:\mathsf{nat})(P\ x)$ will represent the type of functions which associate to each natural number n an object of type $(P\ n)$ and consequently represent proofs of the formula "$\forall x.P(x)$".

4.1.1 Sorts

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

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

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

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

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

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

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

The user will never 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 Type :Type.

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

4.1.2 Constants

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

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

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

4.1.3 Terms

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

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

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

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

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

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

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

4.2 Typed terms

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

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

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

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

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

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

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

Typing rules. In the following, we assume E is a valid environment wrt to inductive definitions. We define simultaneously two judgments. The first one $E[\Gamma] \vdash t : T$ means the term t is well-typed and has type T in the environment E and context Γ. The second judgment $\mathcal{WF}(E)[\Gamma]$ means that the environment E is well-formed and the context Γ is a valid context in this environment. It also means a third property which makes sure that any constant in E was defined in an environment which is included in Γ [1].

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

W-E

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

W-S

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

Def

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

Ax

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

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

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

Var

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

Const

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

Prod

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

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

Lam

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

App

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

Let

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

Remark. We may have $[x := t]u$ well-typed without having $([x : T]u\ t)$ well-typed (where T is a type of t). This is because the value t associated to x may be used in a conversion rule (see section 4.3).

4.3 Conversion rules

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

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

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

According to β-reduction, terms of the *Calculus of Inductive Constructions* enjoy some fundamental properties such as confluence, strong normalization, subject reduction. These results are theoretically of great importance but we will not detail them here and refer the interested reader to [17].

ι-reduction. A specific conversion rule is associated to the inductive objects in the environment. We shall give later on (section 4.5.4) the precise rules but it just says that a destructor applied to an object built from a constructor behaves as expected. This reduction is called ι-reduction and is more precisely studied in [95, 107].

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

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

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

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

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

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

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

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

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

The conversion rule is now exactly:

Conv

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

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

$$E[\Gamma] \vdash [x:T](t\ x) \triangleright t$$

Indeed, as x doesn't occur free in t, for any u one applies to $[x:T](t\ x)$, it β-reduces to $(t\ u)$. So $[x:T](t\ x)$ and t can be identified.

Remark: The η-reduction is not taken into account in the convertibility rule of Coq.

Normal form. A term which cannot be any more reduced is said to be in *normal form*. There are several ways (or strategies) to apply the reduction rule. Among them, we have to mention the *head reduction* which will play an important role (see chapter 7). Any term can be written as $[x_1:T_1]\dots[x_k:T_k](t_0\ t_1\dots t_n)$ where t_0 is not an application. We say then that t_0 is the *head of* t. If we assume that t_0 is $[x:T]u_0$ then one step of β-head reduction of t is:

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

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

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

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

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

4.4 Derived rules for environments

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

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

Substitution Property:

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

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

Abstracting property:

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

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

Pruning property:

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

4.5 Inductive Definitions

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

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

4.5.1 Representing an inductive definition

Inductive definitions without parameters

As for constants, inductive definitions can be defined in a non-empty context.
We write $\mathsf{Ind}(\Gamma)(\ \Gamma_I := \Gamma_C\ )$ an inductive definition valid in a context Γ, a context of definitions Γ_I and a context of constructors Γ_C.

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

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

In a context with a variable A : Set, the lists of elements in A is represented by:

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

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

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

$$\frac{\mathsf{Ind}(\Gamma)(\;\Gamma_I := \Gamma_C\;)\;\in E \quad i = 1..n}{(c_i : C_i\{I_j/I_j\}_{j=1...k}) \in E}$$

Inductive definitions with parameters

We have to slightly complicate the representation above in order to handle the delicate problem of parameters. Let us explain that on the example of List. As they were defined above, the type List can only be used in an environment where we have a variable A : Set. Generally one want to consider lists of elements in different types. For constants this is easily done by abstracting the value over the parameter. In the case of inductive definitions we have to handle the abstraction over several objects.

One possible way to do that would be to define the type List inductively as being an inductive family of type Set $\rightarrow$ Set:

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

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

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

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

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

Examples The declaration for parameterized lists is:

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

The declaration for the length of lists is:

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

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

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

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

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

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

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

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

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

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

4.5.2 Types of inductive objects

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

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

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

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

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

From now on, we write List_A instead of (List A) and Length_A for (Length A).

4.5.3 Well-formed inductive definitions

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

Definitions A type T is an *arity of sort* s if it converts to the sort s or to a product $(x : T)U$ with U an arity of sort s. (For instance $A \to$ Set or $(A :$ Prop$)A \to$ Prop are arities of sort respectively Set and Prop). A *type of constructor of* I is either a term $(I\ t_1 \ldots\ t_n)$ or $(x : T)C$ with C a *type of constructor of* I.

The type of constructor T will be said to *satisfy the positivity condition* for a constant X in the following cases:

- $T = (X\ t_1 \ldots\ t_n)$ and X does not occur free in any t_i
- $T = (x : T)U$ and X occurs only strictly positively in T and the type U satisfies the positivity condition for X

The constant X *occurs strictly positively* in T in the following cases:

- X does not occur in T
- T converts to $(X\ t_1 \ldots\ t_n)$ and X does not occur in any of t_i
- T converts to $(x : U)V$ and X does not occur in type U but occurs strictly positively in type V
- T converts to $(I\ a_1 \ldots\ a_m\ t_1 \ldots\ t_p)$ where I is the name of an inductive declaration of the form $\mathsf{Ind}(\Gamma)[[p_1 : P_1; \ldots ; p_m : P_m]](\ [I : A] := [c_1 : C_1; \ldots ; c_n : C_n]\ )$ (in particular, it is not mutually defined and it has m parameters) and X does not occur in any of the t_i, and the types of constructor $C_i\{p_j/a_j\}_{j=1\ldots m}$ of I satisfy the imbricated positivity condition for X

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

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

Example X occurs strictly positively in $A \to X$ or $X * A$ or $(\texttt{list}X)$ but not in $X \to A$ or $(X \to A) \to A$ assuming the notion of product and lists were already defined. Assuming X has arity $\texttt{nat} \to \texttt{Prop}$ and $\texttt{ex}$ is inductively defined existential quantifier, the occurrence of X in $(\texttt{ex nat}\ [\texttt{n} : \texttt{nat}](\texttt{X n}))$ is also strictly positive.

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

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

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

providing the following side conditions hold:

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

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

4.5.4 Destructors

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

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

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

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

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

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

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

For instance, in order to prove $(l : \mathsf{List\_A})(\mathsf{Length\_A}\ l\ (\mathsf{length}\ l))$ it is enough to prove:
$(\mathsf{Length\_A}\ \mathsf{nil}\ (\mathsf{length}\ \mathsf{nil}))$ and

$(a : A)(l : \mathsf{List\_A})(\mathsf{Length\_A}\ l\ (\mathsf{length}\ l)) \rightarrow (\mathsf{Length\_A}\ (\mathsf{cons}\ A\ a\ l)\ (\mathsf{length}\ (\mathsf{cons}\ A\ a\ l)))$.

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

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

But this operator is rather tedious to implement and use. We choose in this version of Coq to factorize the operator for primitive recursion into two more primitive operations as was first suggested by Th. Coquand in [20]. One is the definition by case analysis. The second one is a definition by guarded fixpoints.

The `Cases...of ...end` construction.

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

This proof will be denoted by a generic term:

$$<P>\texttt{Cases}\ m\ \texttt{of}\ (c_1\ x_{11}\ ...\ x_{1p_1})\texttt{=>}f_1\ |\ \ldots\ |\ (c_n\ x_{n1}...x_{np_n})\texttt{=>}f_n\ \texttt{end}$$

In this expression, if m is a term built from a constructor $(c_i\ u_1 \ldots u_{p_i})$ then the expression will behave as it is specified with i-th branch and will reduce to f_i where the $x_{i1} \ldots x_{ip_i}$ are replaced by the $u_1 \ldots u_p$ according to the ι-reduction.

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

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

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

$$[l : \mathsf{List\_A}]\!<\![H : \mathsf{List\_A}]\mathsf{bool}\!>\!\mathtt{Cases}\ l\ \mathtt{of}\ \mathsf{nil} \Rightarrow \mathsf{true} \mid (\mathsf{cons}\ a\ m) \Rightarrow \mathsf{false}\ \mathtt{end}$$

Remark. In the system Coq the expression above, can be written without mentioning the dummy abstraction: $<\!\mathsf{bool}\!>\mathtt{Cases}\ l\ \mathtt{of}\ \mathsf{nil} \Rightarrow \mathsf{true} \mid (\mathsf{cons}\ a\ m) \Rightarrow \mathsf{false}\ \mathtt{end}$

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

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

For instance we cannot in general have I has type Prop and P has type $I \to \mathsf{Set}$, 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.

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

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

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

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

Prod

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

Prop

$$[I : \mathsf{Prop}|I \to \mathsf{Prop}]$$

$$\frac{I \text{ is an empty or singleton definition}}{[I : \mathsf{Prop}|I \to \mathsf{Set}]} \qquad \frac{I \text{ is an empty or small singleton definition}}{[I : \mathsf{Prop}|I \to \mathsf{Type}(i)]}$$

Set

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

Type

$$\frac{s \in \{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}(j)\}}{[I : \mathsf{Type}(i)|I \to s]}$$

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

Empty and singleton elimination A *singleton definition* has always an informative content, even if it is a proposition.

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

```
Coq < Print eq_rec.
eq_rec =
[A:Set; x:A; P:(A->Set)](eq_rect A x P)
     : (A:Set; x:A; P:(A->Set))(P x)->(y:A)x=y->(P y)

Coq < Extraction eq_rec.
let eq_rec x x0 x1 =
  x0
```

An empty definition has no constructors, in that case also, elimination on any sort is allowed.

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

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

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

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

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

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

Typing rule. Our very general destructor for inductive definition enjoys the following typing rule (we write $<P>$`Cases` c `of` $[x_{11} : T_{11}] \dots [x_{1p_1} : T_{1p_1}]g_1 \dots [x_{n1} : T_{n1}] \dots [x_{np_n} : T_{np_n}]g_n$ `end` for $<P>$`Cases` c `of` $(c_1\ x_{11}\ ...\ x_{1p_1})$`=>`$g_1 \mid \dots \mid (c_n\ x_{n1} ... x_{np_n})$`=>`$g_n$ `end`):

Cases

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

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

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

$$\frac{l : \mathsf{List\_A} \quad P : \mathsf{List\_A} \to s \quad f_1 : (P\ (\mathsf{nil}\ A)) \quad f_2 : (a : A)(l : \mathsf{List\_A})(P\ (\mathsf{cons}\ A\ a\ l))}{<P>\texttt{Cases}\ l\ \texttt{of}\ f_1\ f_2\ \texttt{end} : (P\ l)}$$

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

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

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

$$<P>\texttt{Cases}\ (c_{p_i}\ q_1 \dots q_r\ a_1 \dots a_m)\ \texttt{of}\ f_1 \dots f_l\ \texttt{end}$$

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

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

$$<P>\texttt{Cases}\ (c_{p_i}\ q_1 \dots q_r\ a_1 \dots a_m)\ \texttt{of}\ f_1 \dots f_n\ \texttt{end} \triangleright_\iota (f_i\ a_1 \dots a_m)$$

4.5.5 Fixpoint definitions

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

$$\texttt{Fix}\ \{f_1 : A_1 := t_1 \dots f_n : A_n := t_n\}$$

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

Typing rule

The typing rule is the expected one for a fixpoint.

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

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

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

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

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

can be represented by the term:

$$[P : \mathsf{nat} \to \mathsf{Prop}][f : (P\ \mathsf{O})][g : (n : \mathsf{nat})(P\ n) \to (P\ (\mathsf{S}\ n))]$$
$$\texttt{Fix}\, h\{h : (n : \mathsf{nat})(P\ n) := [n : \mathsf{nat}]{<}P{>}\texttt{Cases}\ n\ \texttt{of}\ f\ [p : \mathsf{nat}](g\ p\ (h\ p))\ \texttt{end}\}$$

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 [54].

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

For doing this the syntax of fixpoints is extended and becomes

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

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

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

The definition of being structurally smaller is a bit technical. One needs first to define the notion of *recursive arguments of a constructor*. For an inductive definition $\mathsf{Ind}(\Gamma)[\Gamma_P](\ \Gamma_I := \Gamma_C\ )$, the type of a constructor c have the form $(p_1 : P_1) \dots (p_r : P_r)(x_1 : T_1) \dots (x_r : T_r)(I_j\ p_1 \dots p_r\ t_1 \dots t_s)$ the recursive arguments will correspond to T_i in which one of the I_l occurs.

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

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

The following definitions are correct, we enter them using the `Fixpoint` command as described in section 1.3.4 and show the internal representation.

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

Coq < Print plus.
plus =
Fix plus
  {plus [n:nat] : nat->nat :=
     [m:nat]Cases n of
              O => m
            | (S p) => (S (plus p m))
            end}
     : nat->nat->nat

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

Coq < Print lgth.
lgth =
Fix lgth
  {lgth [A:Set; l:(list A)] : nat :=
     Cases l of
       nil => O
     | (cons _ l') => (S (lgth A l'))
     end}
     : (A:Set)(list A)->nat

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

Coq < Print sizet.
sizet =
Fix sizet
  {sizet [t:tree] : nat := Cases t of (node f) => (S (sizef f)) end
   with sizef [f:forest] : nat :=
     Cases f of
       emptyf => O
     | (consf t f0) => (plus (sizet t) (sizef f0))
     end}
     : tree->nat
```

Reduction rule

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

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

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

We can illustrate this behavior on examples.

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

  ============================
   (n,m:nat)(plus (S n) m)=(S (plus n m))

Coq < Reflexivity.
Subtree proved!

Coq < Abort.
Current goal aborted

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

  ============================
   (f:forest)(sizet (node f))=(S (sizef f))

Coq < Reflexivity.
Subtree proved!

Coq < Abort.
Current goal aborted
```

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

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

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

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

  ============================
   (t:tree)(sizet t)=(S (sizef (sont t)))

Coq < (** this one fails **)
Coq < Reflexivity.
Error: Impossible to unify (S (sizef (sont t))) with (sizet t)

Coq < Destruct t.
1 subgoal

  t : tree
  ============================
   (f:forest)(sizet (node f))=(S (sizef (sont (node f))))

Coq < Reflexivity.
Subtree proved!
```

Mutual induction

The principles of mutual induction can be automatically generated using the `Scheme` command described in section 7.14.

4.6 Coinductive types

The implementation contains also coinductive definitions, which are types inhabited by infinite objects. More information on coinductive definitions can be found in [55, 56].

Part II

The proof engine

Chapter 5

Vernacular commands

5.1 Displaying

5.1.1 `Print` *qualid* `.`

This command displays on the screen informations about the declared or defined object referred by *qualid*.

Error messages:

1. *qualid* `not a defined object`

Variants:

1. `Print Proof` *qualid* `.`
   This is a synonym to `Print` *qualid* when *qualid* denotes a global constant.

5.1.2 `Print All.`

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

Variants:

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

2. `Print Section` *ident* `.`
   should correspond to a currently open section, this command displays the objects defined since the beginning of this section.

3. `Print.`
   This command displays the axioms and variables declarations in the environment as well as the constants defined since the last variable was introduced.

5.2 Requests to the environment

5.2.1 `Check` *term* `.`

This command displays the type of *term*. When called in proof mode, the term is checked in the local context of the current subgoal.

Variants:

1. `Check` *num term*
   Displays the type of *term* in the context of the *num*-th subgoal.

5.2.2 `Eval` *convtactic* `in` *term* `.`

This command performs the specified reduction on *term*, and displays the resulting term with its type. The term to be reduced may depend on hypothesis introduced in the first subgoal (if a proof is in progress).

Variants:

1. `Eval` *num convtactic* `in` *term* `.`
   Evaluates *term* in the context of the *num*-th subgoal.

See also: section 7.5.

5.2.3 `Extraction` *term* `.`

This command displays the extracted term from *term*. The extraction is processed according to the distinction between Set and Prop; that is to say, between logical and computational content (see section 4.1.1). The extracted term is displayed in Objective Caml syntax, where global identifiers are still displayed as in Coq terms.

Variants:

1. `Recursive Extraction` $qualid_1$ `...` $qualid_n$ `.`
   Recursively extracts all the material needed for the extraction of globals $qualid_1$ ... $qualid_n$.

See also: chapter 17.

5.2.4 `Opaque` $qualid_1$ `...` $qualid_n$ `.`

This command tells not to unfold the the constants $qualid_1 \dots qualid_n$ in tactics using δ-conversion. Unfolding a constant is replacing it by its definition. `Opaque` can only apply on constants originally defined as `Transparent`.

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

See also: sections 7.5, 7.11, 6.1.3

Error messages:

1. `The reference` *qualid* `was not found in the current environment`
   There is no constant referred by *qualid* in the environment. Nevertheless, if you asked `Opaque foo bar` and if `bar` does not exist, `foo` is set opaque.

5.2.5 `Transparent` *qualid*$_1$ ... *qualid*$_n$.

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

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

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

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

Error messages:

1. `The reference` *qualid* `was not found in the current environment`
   There is no constant referred by *qualid* in the environment.

See also: sections 7.5, 7.11, 6.1.3

5.2.6 `Search` *qualid* .

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

1. `The reference` *qualid* `was not found in the current environment`
   There is no constant in the environment named *qualid*.

5.2.7 `SearchPattern` *term* .

This command displays the name and type of all theorems of the current context whose statement's conclusion matches the expression *term* where holes in the latter are denoted by "`?`".

```
Coq < Require Arith.

Coq < SearchPattern (plus ? ?)=?.
le_plus_minus_r: (n,m:nat)(le n m)->(plus n (minus m n))=m
mult_acc_aux: (n,s,m:nat)(plus s (mult n m))=(mult_acc s m n)
plus_sym: (n,m:nat)(plus n m)=(plus m n)
plus_Snm_nSm: (n,m:nat)(plus (S n) m)=(plus n (S m))
plus_assoc_l: (n,m,p:nat)(plus n (plus m p))=(plus (plus n m) p)
plus_permute: (n,m,p:nat)(plus n (plus m p))=(plus m (plus n p))
plus_assoc_r: (n,m,p:nat)(plus (plus n m) p)=(plus n (plus m p))
plus_permute_2_in_4:
  (a,b,c,d:nat)
   (plus (plus a b) (plus c d))=(plus (plus a c) (plus b d))
```

```
plus_tail_plus: (n,m:nat)(plus n m)=(tail_plus n m)
plus_O_n: (n:nat)(plus (0) n)=n
plus_Sn_m: (n,m:nat)(plus (S n) m)=(S (plus n m))
mult_n_Sm: (n,m:nat)(plus (mult n m) n)=(mult n (S m))
```

Patterns need not be linear: you can express that the same expression must occur in two places by using indexed '?".

```
Coq < Require Arith.

Coq < SearchPattern (plus ?1 ?)=?1.
```

5.2.8 `SearchRewrite` *term*.

This command displays the name and type of all theorems of the current context whose statement's conclusion is an equality of which one side matches the expression *term*=. Holes in *term* are denoted by "?".

```
Coq < Require Arith.

Coq < SearchRewrite (plus ? ?).
le_plus_minus: (n,m:nat)(le n m)->m=(plus n (minus m n))
le_plus_minus_r: (n,m:nat)(le n m)->(plus n (minus m n))=m
mult_plus_distr:
  (n,m,p:nat)(mult (plus n m) p)=(plus (mult n p) (mult m p))
mult_plus_distr_r:
  (n,m,p:nat)(mult n (plus m p))=(plus (mult n m) (mult n p))
mult_acc_aux: (n,s,m:nat)(plus s (mult n m))=(mult_acc s m n)
plus_sym: (n,m:nat)(plus n m)=(plus m n)
plus_Snm_nSm: (n,m:nat)(plus (S n) m)=(plus n (S m))
plus_assoc_l: (n,m,p:nat)(plus n (plus m p))=(plus (plus n m) p)
plus_permute: (n,m,p:nat)(plus n (plus m p))=(plus m (plus n p))
plus_assoc_r: (n,m,p:nat)(plus (plus n m) p)=(plus n (plus m p))
plus_permute_2_in_4:
  (a,b,c,d:nat)
   (plus (plus a b) (plus c d))=(plus (plus a c) (plus b d))
plus_tail_plus: (n,m:nat)(plus n m)=(tail_plus n m)
plus_n_O: (n:nat)n=(plus n (0))
plus_O_n: (n:nat)(plus (0) n)=n
plus_n_Sm: (n,m:nat)(S (plus n m))=(plus n (S m))
plus_Sn_m: (n,m:nat)(plus (S n) m)=(S (plus n m))
mult_n_Sm: (n,m:nat)(plus (mult n m) n)=(mult n (S m))
```

Variants:

1. `Search` *qualid* `inside` $module_1 \ldots module_n$.
   `SearchPattern` *term* `inside` $module_1 \ldots module_n$.
   `SearchRewrite` *term* `inside` $module_1 \ldots module_n$.

   This restricts the search to constructions defined in modules $module_1 ... module_n$.

2. `Search` *qualid* `outside` $module_1 \ldots module_n$.
   `SearchPattern` *term* `outside` $module_1 \ldots module_n$.
   `SearchRewrite` *term* `outside` $module_1 \ldots module_n$.

   This restricts the search to constructions not defined in modules $module_1 ... module_n$.

Error messages:

1. `Module/section` *module* `not found` No module *module* has been required (see section 5.4.2).

5.2.9 `Locate` *qualid* `.`

This command displays the full name of the qualified identifier *qualid* and consequently the Coq module in which it is defined.

```
Coq < Locate nat.
Coq.Init.Datatypes.nat

Coq < Locate Datatypes.O.
Coq.Init.Datatypes.O

Coq < Locate Init.Datatypes.O.
Coq.Init.Datatypes.O

Coq < Locate Coq.Init.Datatypes.O.
Coq.Init.Datatypes.O

Coq < Locate I.Dont.Exist.
Error: I.Dont.Exist is not a defined object
```

5.3 Loading files

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

5.3.1 `Load` *ident* `.`

This command loads the file named *ident*`.v`, searching successively in each of the directories specified in the *loadpath*. (see section 5.5)

Variants:

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

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

Error messages:

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

5.4 Compiled files

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

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

5.4.1 `Read Module` *qualid* `.`

This looks for a physical file `file.vo` mapped to logical name *qualid* in the current Coq loadpath, then loads its contents but does not open it: its contents remains accesible to the user only by using names prefixed by the module name (i.e. *qualid* or any other qualified names denoting the same module).

5.4.2 `Require` *dirpath* `.`

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

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

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

Variants:

1. `Require Export` *qualid* `.`
   This command acts as `Require` *qualid*. But if a module *A* contains a command `Require Export` *B*, then the command `Require` *A* opens the module *B* as if the user would have typed `Require`*B*.

2. `Require` *qualid* *string* `.`
   Specifies the file to load as being *string* but containing module *qualid* which is then opened.

These different variants can be combined.

Error messages:

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

2. `Can't find module toto on loadpath`
   The command did not find the file `toto.vo`. Either `toto.v` exists but is not compiled or `toto.vo` is in a directory which is not in your `LoadPath` (see section 5.5).

3. `Bad magic number`
   The file *ident*`.vo` was found but either it is not a Coq compiled module, or it was compiled with an older and incompatible version of Coq.

See also: chapter 11

5.4.3 `Print Modules.`

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

5.4.4 `Declare ML Module` *string*$_1$ `..` *string*$_n$ `.`

This commands loads the Objective Caml compiled files *string*$_1$ ... *string*$_n$ (dynamic link). It is mainly used to load tactics dynamically. The files are searched into the current Objective Caml loadpath (see the command `Add ML Path` in the section 5.5). Loading of Objective Caml files is only possible under the bytecode version of `coqtop` (i.e. `coqtop` called with options `-byte`, see chapter 11). **Error messages:**

1. `File not found on loadpath :` *string*
2. `Loading of ML object file forbidden in a native Coq`

5.4.5 `Print ML Modules.`

This print the name of all Objective Caml modules loaded with `Declare ML Module`. To know from where these module were loaded, the user should use the command `Locate File` (see page 100)

5.5 Loadpath

There are currently two loadpaths in Coq. A loadpath where seeking Coq files (extensions `.v` or `.vo` or `.vi`) and one where seeking Objective Caml files. The default loadpath contains the directory "`.`" denoting the current directory and mapped to the empty logical path (see section 2.6).

5.5.1 `Pwd.`

This command displays the current working directory.

5.5.2 `Cd` *string* `.`

This command changes the current directory according to *string* which can be any valid path.

Variants:

1. `Cd.`
   Is equivalent to `Pwd.`

5.5.3 `Add LoadPath` *string* `as` *dirpath* `.`

This command adds the path *string* to the current Coq loadpath and maps it to the logical directory *dirpath*, which means that every file `M.v` physically lying in directory *string* becomes accessible through logical name "*dirpath*`.M`".

Remark: `Add LoadPath` also adds *string* to the current ML loadpath.

Variants:

1. `Add LoadPath` *string* `.`
   Performs as `Add LoadPath` *string* `as` *dirpath* but for the empty directory path.

5.5.4 `Add Rec LoadPath` *string* `as` *dirpath* `.`

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

Remark: `Add Rec LoadPath` also recursively adds *string* to the current ML loadpath.

Variants:

1. `Add Rec LoadPath` *string* `.`
   Works as `Add Rec LoadPath` *string* `as` *dirpath* but for the empty logical directory path.

5.5.5 `Remove LoadPath` *string* `.`

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

5.5.6 `Print LoadPath.`

This command displays the current Coq loadpath.

5.5.7 `Add ML Path` *string* `.`

This command adds the path *string* to the current Objective Caml loadpath (see the command `Declare ML Module` in the section 5.4).

Remark: This command is implied by `Add LoadPath` *string* `as` *dirpath*.

5.5.8 `Add Rec ML Path` *string* `.`

This command adds the directory *string* and all its subdirectories to the current Objective Caml loadpath (see the command `Declare ML Module` in the section 5.4).

Remark: This command is implied by `Add Rec LoadPath` *string* `as` *dirpath*.

5.5.9 `Print ML Path` *string* `.`

This command displays the current Objective Caml loadpath. This command makes sense only under the bytecode version of `coqtop`, i.e. using option `-byte` (see the command `Declare ML Module` in the section 5.4).

5.5.10 `Locate File` *string* `.`

This command displays the location of file *string* in the current loadpath. Typically, *string* is a `.cmo` or `.vo` or `.v` file.

5.5.11 `Locate Library` *dirpath* `.`

This command gives the status of the Coq module *dirpath*. It tells if the module is loaded and if not searches in the load path for a module of logical name *dirpath*.

5.6 States and Reset

5.6.1 `Reset` *ident* `.`

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

Error messages:

1. *ident*: `no such entry`

5.6.2 `Back.`

This commands undoes all the effects of the last vernacular command. This does not include commands that only access to the environment like those described in the previous sections of this chapter (for instance `Require` and `Load` can be undone, but not `Check` and `Locate`). Commands read from a vernacular file are considered as a single command.

Variants:

1. `Back` n
   Undoes n vernacular commands.

Error messages:

1. `Reached begin of command history`
   Happens when there is vernacular command to undo.

5.6.3 `Restore State` *ident* `.`

Restores the state contained in the file *string*.

Variants:

1. `Restore State` *ident*
   Equivalent to `Restore State "`*ident*`.coq"..`

2. `Reset Initial.`
   Goes back to the initial state (like after the command `coqtop`).

5.6.4 `Write State` *string*`.`

Writes the current state into a file *string* for use in a further session. This file can be given as the `inputstate` argument of the commands `coqtop` and `coqc`.

Variants:

1. `Write State` *ident*
   Equivalent to `Write State "`*ident*`.coq".`. The state is saved in the current directory (see 99).

5.7 Syntax facilities

We present quickly in this section some syntactic facilities. We will only sketch them here and refer the interested reader to chapter 9 for more details and examples.

5.7.1 `Set Implicit Arguments.`

This command sets the implicit argument mode on. Under this mode, the arguments of declared constructions (constants, inductive types, variables, ...) which can automatically be deduced from the others arguments (typically type arguments in polymorphic functions) are skipped. They are not printed and the user must not give them. To show what are the implicit arguments associated to a declaration *qualid*, use `Print` *qualid*. You can change the implicit arguments of a specific declaration by using the command `Implicits` (see section 5.7.2). You can explicitely give an argument which otherwise should be implicit by using the symbol `!` (see section 2.7.1).

To set the implicit argument mode off, use `Unset Implicit Arguments.`

Variants:

1. `Implicit Arguments On.`
   This is a deprecated equivalent to `Set Implicit Arguments.`

2. `Implicit Arguments Off.`
   This is a deprecated equivalent to `Unset Implicit Arguments.`

See also: section 2.7.1

5.7.2 `Implicits` *qualid* `[`

$num_1 \dots num_n$ `]`

This sets the implicit arguments of reference *qualid* to be the arguments at positions num_1 $\dots num_n$. As a particular case, if the list of numbers is empty then no implicit argument is associated to *qualid*.

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

This command defines *ident* as an abbreviation with implicit arguments. Implicit arguments are denoted in *term* by `?` and they will have to be synthesized by the system.

Remark: Since it may contain don't care variables `?`, the argument *term* cannot be typechecked at definition time. But each of its subsequent usages will be.

See also: section 2.7.2

5.7.4 `Syntax` *ident* *syntax-rules*.

This command addresses the extensible pretty-printing mechanism of Coq. It allows $ident_2$ to be pretty-printed as specified in *syntax-rules*. Many examples of the `Syntax` command usage may be found in the `PreludeSyntax` file (see directory `$COQLIB/theories/INIT`).

See also: chapter 9

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

This command allows to give explicitly new grammar rules for parsing the user's own notation. It may be used instead of the `Syntactic Definition` pragma. It can also be used by an advanced Coq's user who programs his own tactics.

See also: chapters 9

5.7.6 `Infix` *num* *string* *qualid* .

This command declares the prefix operator denoted by *qualid* as infix, with the syntax *term string term*. *num* is the precedence associated to the operator; it must lie between 1 and 10. The infix operator *string* associates to the left. *string* must be a legal token. Both grammar and pretty-print rules are automatically generated for *string*.

Variants:

1. `Infix` *assoc* *num* *string* *qualid* .
   Declares the full names denoted by *qualid* as an infix operator with an alternate associativity. *assoc* may be one of `LEFTA`, `RIGHTA` and `NONA`. The default is `LEFTA`. When an associativity is given, the precedence level must lie between 6 and 9.

5.8 Miscellaneous

5.8.1 `Quit.`

This command permits to quit Coq.

5.8.2 `Drop.`

This is used mostly as a debug facility by Coq's implementors and does not concern the casual user. This command permits to leave Coq temporarily and enter the Objective Caml toplevel. The Objective Caml command:

```
#use "include";;
```

add the right loadpaths and loads some toplevel printers for all abstract types of Coq- section_path, identfifiers, terms, judgements, ... You can also use the file `base_include` instead, that loads only the pretty-printers for section_paths and identifiers. You can return back to Coq with the command:

```
go();;
```

Warnings:

1. It only works with the bytecode version of Coq (i.e. `coqtop` called with option `-byte`, see page 203).
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 11.4).

5.8.3 `Set Silent.`

This command turns off the normal displaying.

5.8.4 `Unset Silent.`

This command turns the normal display on.

5.8.5 `Time` *command* `.`

This command executes the vernac command *command* and display the time needed to execute it.

Chapter 6

Proof handling

In Coq's proof editing mode all top-level commands documented in chapter 5 remain available and the user has access to specialized commands dealing with proof development pragmas documented in this section. He can also use some other specialized commands called *tactics*. They are the very tools allowing the user to deal with logical reasoning. They are documented in chapter 7. When switching in editing proof mode, the prompt `Coq <` is changed into *ident* `<` where *ident* is the declared name of the theorem currently edited.

At each stage of a proof development, one has a list of goals to prove. Initially, the list consists only in the theorem itself. After having applied some tactics, the list of goals contains the subgoals generated by the tactics.

To each subgoal is associated a number of hypotheses we call the *local context* of the goal. Initially, the local context is empty. It is enriched by the use of certain tactics (see mainly section 7.3.5).

When a proof is achieved the message `Subtree proved!` is displayed. One can then store this proof as a defined constant in the environment. Because there exists a correspondence between proofs and terms of λ-calculus, known as the *Curry-Howard isomorphism* [62, 5, 59, 64], Coq stores proofs as terms of CIC. Those terms are called *proof terms*.

It is possible to edit several proofs at the same time: see section 6.1.7

Error message: When one attempts to use a proof editing command out of the proof editing mode, Coq raises the error message : `No focused proof`.

6.1 Switching on/off the proof editing mode

6.1.1 `Goal` *form* .

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

Error messages:

1. `the term` *form* `has type ... which should be Set, Prop or Type`
2. `repeated goal not permitted in refining mode` the command `Goal` cannot be used while a proof is already being edited.

See also: section 6.1.3

6.1.2 `Qed.`

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

Error messages:

1. `Attempt to save an incomplete proof`
2. *ident* `already exists`
   The implicit name is already defined. You have then to provide explicitly a new name (see variant 2 below).
3. Sometimes an error occurs when building the proof term, because tactics do not enforce completely the term construction constraints.

   The user should also be aware of the fact that since the proof term is completely rechecked at this point, one may have to wait a while when the proof is large. In some exceptional cases one may even incur a memory overflow.

Variants:

1. `Defined.`
   Defines the proved term as a transparent constant.
2. `Save.`
   Is equivalent to `Qed`.
3. `Save` *ident* `.`
   Forces the name of the original goal to be *ident*. This command (and the following ones) can only be used if the original goal has been opened using the `Goal` command.
4. `Save Theorem` *ident* `.`
   `Save Lemma` *ident* `.`
   Are equivalent to `Save` *ident* `.`
5. `Save Remark` *ident* `.`
   Defines the proved term as a constant that will not be accessible without using a qualified name after the end of the current section.
6. `Save Fact` *ident* `.`
   Defines the proved term as a constant that will not be accessible without using a qualified name after the closing of two levels of sectioning.

6.1.3 `Theorem` *ident* `:` *form* `.`

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

Error message: (see section 6.1.1)

Variants:

1. `Lemma` *ident* `:` *form* `.`
   It is equivalent to `Theorem` *ident* `:` *form* `.`

2. `Remark` *ident* `:` *form* `.`
   Analogous to `Theorem` except that *ident* will be accessible only by a qualified name after closing the current section.

3. `Fact` *ident* `:` *form* `.`
   Analogous to `Theorem` except that *ident* is accessible by a short name after closing the current section but will be accessible only by a qualified name after closing the section which is above the current section.

4. `Definition` *ident* `:` *form* `.`
   Analogous to `Theorem`, intended to be used in conjunction with `Defined` (see 1) in order to define a transparent constant.

5. `Local` *ident* `:` *form* `.`
   Analogous to `Definition` except that the definition is turned into a local definition on objects depending on it after closing the current section.

6.1.4 `Proof` *term* `.`

This command applies in proof editing mode. It is equivalent to `Exact` *term* `; Save.` That is, you have to give the full proof in one gulp, as a proof term (see section 7.2.1).

Variants:

1. `Proof.` is a noop which is useful to delimit the sequence of tactic commands which start a proof, after a `Theorem` command. It is a good practice to use `Proof.` as an opening parenthesis, closed in the script with a closing `Qed.`

6.1.5 `Abort.`

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

Error messages:

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

Variants:

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

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

6.1.6 `Suspend.`

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

6.1.7 `Resume.`

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

Error messages:

1. `No proof-editing in progress`

Variants:

1. `Resume` *ident*`.`
   Restarts the editing of the proof named *ident*. This can be used to navigate between currently edited proofs.

Error messages:

1. `No such proof`

6.2 Navigation in the proof tree

6.2.1 `Undo.`

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

Error messages:

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

Variants:

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

6.2.2 `Set Undo` *num*`.`

This command changes the maximum number of `Undo`'s that will be possible when doing a proof. It only affects proofs started after this command, such that if you want to change the current undo limit inside a proof, you should first restart this proof.

6.2.3 `Unset Undo.`

This command resets the default number of possible `Undo` commands (which is currently 12).

6.2.4 `Restart.`

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

Error messages:

1. `No focused proof to restart`

6.2.5 `Focus.`

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

6.2.6 `Unfocus.`

Turns off the focus mode.

6.3 Displaying information

6.3.1 `Show.`

This command displays the current goals.

Variants:

1. `Show` *num* `.`
   Displays only the *num*-th subgoal.
   Error messages:

   (a) `No such goal`
   (b) `No focused proof`

2. `Show Implicits.`
   Displays the current goals, printing the implicit arguments of constants.

3. `Show Implicits` *num* `.`
   Same as above, only displaying the *num*-th subgoal.

4. `Show Script.`
   Displays the whole list of tactics applied from the beginning of the current proof. This tactics script may contain some holes (subgoals not yet proved). They are printed under the form `<Your Tactic Text here>`.

5. `Show Tree.`
   This command can be seen as a more structured way of displaying the state of the proof than that provided by `Show Script`. Instead of just giving the list of tactics that have been applied, it shows the derivation tree constructed by then. Each node of the tree contains the conclusion of the corresponding sub-derivation (i.e. a goal with its corresponding local context) and the tactic that has generated all the sub-derivations. The leaves of this tree are the goals which still remain to be proved.

6. `Show Proof.`
   It displays the proof term generated by the tactics that have been applied. If the proof is not completed, this term contain holes, which correspond to the sub-terms which are still to be constructed. These holes appear as a question mark indexed by an integer, and applied to the list of variables in the context, since it may depend on them. The types obtained by abstracting away the context from the type of each hole-placer are also printed.

7. `Show Conjectures.`
   It prints the list of the names of all the theorems that are currently being proved. As it is possible to start proving a previous lemma during the proof of a theorem, this list may contain several names.

8. `Show Intro.`
   If the current goal begins by at least one product, this command prints the name of the first product, as it would be generated by an anonymous `Intro`. The aim of this command is to ease the writing of more robust scripts. For example, with an appropriate Proof General macro, it is possible to transform any anonymous `Intro` into a qualified one such as `Intro y13`. In the case of a non-product goal, it prints nothing.

9. `Show Intros.`
   This command is similar to the previous one, it simulates the naming process of an `Intros`.

6.3.2 `Set Hyps_limit` *num* `.`

This command sets the maximum number of hypotheses displayed in goals after the application of a tactic. All the hypotheses remains usable in the proof development.

6.3.3 `Unset Hyps_limit.`

This command goes back to the default mode which is to print all available hypotheses.

Chapter 7

Tactics

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

Each (sub)goal is denoted with a number. The current goal is numbered 1. By default, a tactic is applied to the current goal, but one can address a particular goal in the list by writing *n:tactic* which means *"apply tactic tactic to goal number n"*. We can show the list of subgoals by typing `Show` (see section 6.3.1).

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

Tactics are build from tacticals and atomic tactics. There are, at least, three levels of atomic tactics. The simplest one implements basic rules of the logical framework. The second level is the one of *derived rules* which are built by combination of other tactics. The third one implements heuristics or decision procedures to build a complete proof of a goal.

7.1 Syntax of tactics and tacticals

A tactic is applied as an ordinary command. If the tactic does not address the first subgoal, the command may be preceded by the wished subgoal number. See figure 7.1 for the syntax of tactic invocation and tacticals.

Remarks:

1. The infix tacticals `Orelse` and "… `;` …" are associative. The tactical `Orelse` binds more than the prefix tacticals `Try`, `Repeat`, `Do`, `Info` and `Abstract` which themselves bind more than the postfix tactical "`... ;[ ... ]`" which binds more than "… `;` …".

   For instance

   `Try Repeat` $tactic_1$ `Orelse` $tactic_2$`;`$tactic_3$`;[`$tactic_{31}$`|...|`$tactic_{3n}$`];`$tactic_4$`.`

tactic	::=	*atomic_tactic*
	|	`(` *tactic* `)`
	|	*tactic* `Orelse` *tactic*
	|	`Repeat` *tactic*
	|	`Do` *num tactic*
	|	`Info` *tactic*
	|	`Try` *tactic*
	|	`First [` *tactic* | `...` | *tactic* `]`
	|	`Solve [` *tactic* | `...` | *tactic* `]`
	|	`Abstract` *tactic*
	|	`Abstract` *tactic* `using` *ident*
	|	*tactic* `;` *tactic*
	|	*tactic* `; [` *tactic* | `...` | *tactic* `]`
tactic_invocation	::=	*num* `:` *tactic* `.`
	|	*tactic* `.`

Figure 7.1: Invocation of tactics and tacticals

is understood as

`(Try (Repeat (`*tactic*$_1$ `Orelse` *tactic*$_2$`))); ((`*tactic*$_3$`;[`*tactic*$_{31}$ | `...` | *tactic*$_{3n}$`]);`*tactic*$_4$`).`

2. An *atomic_tactic* is any of the tactics listed below.

7.2 Explicit proof as a term

7.2.1 `Exact` *term*

This tactic applies to any goal. It gives directly the exact proof term of the goal. Let `T` be our goal, let `p` be a term of type `U` then `Exact p` succeeds iff `T` and `U` are convertible (see section 4.3).

Error messages:

1. `Not an exact proof`

7.2.2 `Refine` *term*

This tactic allows to give an exact proof but still with some holes. The holes are noted "`?`".

Error messages:

1. `invalid argument`: the tactic `Refine` doesn't know what to do with the term you gave.
2. `Refine passed ill-formed term`: the term you gave is not a valid proof (not easy to debug in general). This message may also occur in higher-level tactics, which call `Refine` internally.
3. `There is an unknown subterm I cannot solve`: there is a hole in the term you gave which type cannot be inferred. Put a cast around it.

This tactic is currently given as an experiment. An example of use is given in section 8.1.

7.3 Basics

Tactics presented in this section implement the basic typing rules of CIC given in chapter 4.

7.3.1 `Assumption`

This tactic applies to any goal. It implements the "Var" rule given in section 4.2. It looks in the local context for an hypothesis which type is equal to the goal. If it is the case, the subgoal is proved. Otherwise, it fails.

Error messages:

1. `No such assumption`

7.3.2 `Clear` *ident*`.`

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

Variants:

1. `Clear` $ident_1$ `...`$ident_n$`.`
   This is equivalent to `Clear` $ident_1$`.` `...Clear` $ident_n$`.`

2. `ClearBody` *ident*`.`
   This tactic expects *ident* to be a local definition then clears its body. Otherwise said, this tactic turns a definition into an assumption.

Error messages:

1. `No such assumption`

2. *ident* `is used in the conclusion`

3. *ident* `is used in the hypothesis` *ident'*

7.3.3 `Move` $ident_1$ `after` $ident_2$`.`

This moves the hypothesis named $ident_1$ in the local context after the hypothesis named $ident_2$.

If $ident_1$ comes before $ident_2$ in the order of dependences, then all hypotheses between $ident_1$ and $ident_2$ which (possibly indirectly) depend on $ident_1$ are moved also.

If $ident_1$ comes after $ident_2$ in the order of dependences, then all hypotheses between $ident_1$ and $ident_2$ which (possibly indirectly) occur in $ident_1$ are moved also.

Error messages:

1. `No such assumption:` $ident_i$

2. `Cannot move` $ident_1$ `after` $ident_2$`: it occurs in` $ident_2$

3. `Cannot move` $ident_1$ `after` $ident_2$`: it depends on` $ident_2$

7.3.4 `Rename` *ident*$_1$ `into` *ident*$_2$`.`

This renames hypothesis *ident*$_1$ into *ident*$_2$ in the current context[1]

Error messages:

1. `No such assumption`
2. *ident*$_2$ `is already used`

7.3.5 `Intro`

This tactic applies to a goal which is either a product or starts with a let binder. If the goal is a product, the tactic implements the "Lam" rule given in section 4.2[2]. If the goal starts with a let binder then the tactic implements a mix of the "Let" and "Conv".

If the current goal is a dependent product `(x:T)U` (resp `[x:=t]U`) then `Intro` puts `x:T` (resp `x:=t`) in the local context. The new subgoal is `U`.

If the goal is a non dependent product T -> U, then it puts in the local context either `H`*n*`:T` (if `T` is `Set` or `Prop`) or `X`*n*`:T` (if the type of `T` is `Type`) *n* is such that `H`*n* or `X`*n* `l`*n* or are fresh identifiers. In both cases the new subgoal is `U`.

If the goal is neither a product nor starting with a let definition, the tactic `Intro` applies the tactic `Red` until the tactic `Intro` can be applied or the goal is not reducible.

Error messages:

1. `No product even after head-reduction`
2. *ident* `is already used`

Variants:

1. `Intros`
   Repeats `Intro` until it meets the head-constant. It never reduces head-constants and it never fails.

2. `Intro` *ident*
   Applies `Intro` but forces *ident* to be the name of the introduced hypothesis.

   Error message: `name` *ident* `is already bound`

   Remark: If a name used by `Intro` hides the base name of a global constant then the latter can still be referred to by a qualified name (see 2.6).

3. `Intros` *ident*$_1$ `...` *ident*$_n$
   Is equivalent to the composed tactic `Intro` *ident*$_1$`; ... ; Intro` *ident*$_n$.
   More generally, the `Intros` tactic takes a pattern as argument in order to introduce names for components of an inductive definition or to clear introduced hypotheses; This is explained in 7.7.3.

[1]but it does not rename the hypothesis in the proof-term...
[2]Actually, only the second subgoal will be generated since the other one can be automatically checked.

4. `Intros until` *ident*
   Repeats `Intro` until it meets a premise of the goal having form `(` *ident* `:` *term* `)` and discharges the variable named *ident* of the current goal.

   Error message: `No such hypothesis in current goal`

5. `Intros until` *num*
   Repeats `Intro` until the *num*-th non-dependant premise. For instance, on the subgoal `(x,y:nat)x=y->(z:nat)h=x->z=y` the tactic `Intros until 2` is equivalent to `Intros x y H z H0` (assuming `x, y, H, z` and `H0` do not already occur in context).

   Error message: `No such hypothesis in current goal`
   Happens when *num* is 0 or is greater than the number of non-dependant products of the goal.

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

   Error messages:

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

7. `Intro` $ident_1$ `after` $ident_2$
   Behaves as previously but $ident_1$ is the name of the introduced hypothesis. It is equivalent to `Intro` $ident_1$`; Move` $ident_1$ `after` $ident_2$.

   Error messages:

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

7.3.6 `Apply` *term*

This tactic applies to any goal. The argument *term* is a term well-formed in the local context. The tactic `Apply` tries to match the current goal against the conclusion of the type of *term*. If it succeeds, then the tactic returns as many subgoals as the instantiations of the premises of the type of *term*.

Error messages:

1. `Impossible to unify ... with ...`
   Since higher order unification is undecidable, the `Apply` tactic may fail when you think it should work. In this case, if you know that the conclusion of *term* and the current goal are unifiable, you can help the `Apply` tactic by transforming your goal with the `Change` or `Pattern` tactics (see sections 7.5.7, 7.3.10).

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

Variants:

1. `Apply` *term* `with` $term_1$ `...` $term_n$
   Provides `Apply` with explicit instantiations for all dependent premises of the type of *term* which do not occur in the conclusion and consequently cannot be found by unification. Notice that $term_1 \dots term_n$ must be given according to the order of these dependent premises of the type of *term*.

   Error message: `Not the right number of missing arguments`

2. `Apply` *term* `with` ref_1 `:=` $term_1$ `...` ref_n `:=` $term_n$
   This also provides `Apply` with values for instantiating premises. But variables are referred by names and non dependent products by order (see syntax in the section 7.3.11).

3. `EApply` *term*
   The tactic `EApply` behaves as `Apply` but does not fail when no instantiation are deducible for some variables in the premises. Rather, it turns these variables into so-called existential variables which are variables still to instantiate. An existential variable is identified by a name of the form `?`n where n is a number. The instantiation is intended to be found later in the proof.

   An example of use of `EApply` is given in section 8.2.

4. `LApply` *term*

   This tactic applies to any goal, say `G`. The argument *term* has to be well-formed in the current context, its type being reducible to a non-dependent product `A -> 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: Be careful, when *term* contains more than one non dependent product the tactic `LApply` only takes into account the first product.

7.3.7 `LetTac` *ident* `:=` *term*

This replaces *term* by *ident* in the conclusion and the hypotheses of the current goal and adds the new definition *ident* := *term* to the local context.

Variants:

1. `LetTac` *ident* `:=` *term* `in Goal`

   This is equivalent to the above form but applies only to the conclusion of the goal.

2. `LetTac` $ident_0$ `:=` *term* `in` $ident_1$

   This behaves the same but substitutes *term* not in the goal but in the hypothesis named $ident_1$.

3. `LetTac` $ident_0$ `:=` *term* `in` num_1 `...` num_n $ident_1$

   This notation allows to specify which occurrences of the hypothesis named $ident_1$ (or the goal if $ident_1$ is the word `Goal`) should be substituted. The occurrences are numbered from left to right. A negative occurrence number means an occurrence which should not be substituted.

4. `LetTac` $ident_0$ `:=` *term* `in` num_1^1 `...` $num_{n_1}^1$ $ident_1$ `...`num_1^m `...` $num_{n_m}^m$ $ident_m$

   This is the general form. It substitutes *term* at occurrences $num_1^i \dots num_{n_i}^i$ of hypothesis $ident_i$. One of the *ident*'s may be the word `Goal`.

5. `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.

6. `Pose` *term*
   This behaves as `Pose` *ident* `:=` *term* but *ident* is generated by Coq.

7.3.8 `Assert` *ident* `:` *form*

This tactic applies to any goal. `Assert H : U` adds a new hypothesis of name `H` asserting `U` to the current goal and opens a new subgoal `U`[3]. The subgoal `U` comes first in the list of subgoals remaining to prove.

Error messages:

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

Variants:

1. `Assert` *form*
   This behaves as `Assert` *ident* `:` *form* but *ident* is generated by Coq.

2. `Assert` *ident* `:=` *term*
   This behaves as `Assert` *ident* `:` *type*`;` `[Exact` *term* | `Idtac]` where *type* is the type of *term*.

3. `Cut` *form*
   This tactic applies to any goal. It implements the non dependent case of the "App" rule given in section 4.2. (This is Modus Ponens inference rule.) `Cut U` transforms the current goal `T` into the two following subgoals: `U -> T` and `U`. The subgoal `U -> T` comes first in the list of remaining subgoal to prove.

[3]This corresponds to the cut rule of sequent calculus.

7.3.9 `Generalize` *term*

This tactic applies to any goal. It generalizes the conclusion w.r.t. one subterm of it. For example:

```
Coq < Show.
1 subgoal

  x : nat
  y : nat
  ============================
   (le O (plus (plus x y) y))

Coq < Generalize (plus (plus x y) y).
1 subgoal

  x : nat
  y : nat
  ============================
   (n:nat)(le O n)
```

If the goal is G and t is a subterm of type T in the goal, then `Generalize` t replaces the goal by `(x:`T`)` G' where G' is obtained from G by replacing all occurrences of t by `x`. The name of the variable (here `n`) is chosen accordingly to T.

Variants:

1. `Generalize` *term*$_1$ `...` *term*$_n$
   Is equivalent to `Generalize` *term*$_n$`; ... ; Generalize` *term*$_1$. Note that the sequence of *term*$_i$'s are processed from n to 1.

2. `Generalize Dependent` *term*
   This generalizes *term* but also *all* hypotheses which depend on *term*.

7.3.10 `Change` *term*

This tactic applies to any goal. It implements the rule "Conv" given in section 4.3. `Change U` replaces the current goal `T` with a `U` providing that `U` is well-formed and that `T` and `U` are convertible.

Error messages:

1. `Not convertible`

Variants:

1. `Change` *term* `in` *ident*
   This applies the `Change` tactic not to the goal but to the hypothesis *ident*.

See also: 7.5

7.3.11 Bindings list

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

A bindings list can also be a simple list of terms $term_1$ $term_2$ ...$term_n$. In that case the references to which these terms correspond are determined by the tactic. In case of `Elim` *term* (see section 2) the terms should correspond to all the dependent products in the type of *term* while in the case of `Apply` *term* only the dependent products which are not bound in the conclusion of the type are given.

7.4 Negation and contradiction

7.4.1 `Absurd` *term*

This tactic applies to any goal. The argument *term* is any proposition `P` of type `Prop`. This tactic applies `False` elimination, that is it deduces the current goal from `False`, and generates as subgoals `~P` and `P`. It is very useful in proofs by cases, where some cases are impossible. In most cases, `P` or `~P` is one of the hypotheses of the local context.

7.4.2 `Contradiction`

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

Error messages:

1. `No such assumption`

7.5 Conversion tactics

This set of tactics implements different specialized usages of the tactic `Change`.

7.5.1 `Cbv` $\textbf{\textit{flag}}_1$ ... $\textbf{\textit{flag}}_n$, `Lazy` $\textbf{\textit{flag}}_1$ ... $\textbf{\textit{flag}}_n$ and `Compute`

These parameterized reduction tactics apply to any goal and perform the normalization of the goal according to the specified flags. Since the reduction considered in Coq include β (reduction of functional application), δ (unfolding of transparent constants, see 5.2.5), ι (reduction of `Cases`, `Fix` and `CoFix` expressions) and ζ (removal of local definitions), every flag is one of `Beta`, `Delta`, `Iota`, `Zeta`, `[`$qualid_1$...$qualid_k$`]` and `-[`$qualid_1$...$qualid_k$`]`. The last two flags give the list of constants to unfold, or the list of constants not to unfold. These two flags can occur only after the `Delta` flag. For these tactics, the `Delta` flag does not apply to variables bound by a let-in construction of which the unfolding is controlled by the `Zeta` flag only. In addition, there is a flag `Evar` to perform instantiation of exitential variables ("?") when an instantiation actually exists. The goal may be normalized with two strategies: *lazy* (`Lazy` tactic), or *call-by-value* (`Cbv` tactic).

Notice that, for these tactics, the `Delta` flag without rest should be understood as unfolding all The lazy strategy is a call-by-need strategy, with sharing of reductions: the arguments of a function call are partially evaluated only when necessary, but if an argument is used several times, it is computed only once. This reduction is efficient for reducing expressions with dead code. For instance, the proofs of a proposition $\exists_T\ x.P(x)$ reduce to a pair of a witness t, and a proof that t verifies the predicate P. Most of the time, t may be computed without computing the proof of $P(t)$, thanks to the lazy strategy.

The call-by-value strategy is the one used in ML languages: the arguments of a function call are evaluated first, using a weak reduction (no reduction under the λ-abstractions). Despite the lazy strategy always performs fewer reductions than the call-by-value strategy, the latter should be preferred for evaluating purely computational expressions (i.e. with few dead code).

Variants:

1. `Compute`
   This tactic is an alias for `Cbv Beta Delta Evar Iota Zeta`.

Error messages:

1. `Delta must be specified before`
   A list of constants appeared before the `Delta` flag.

7.5.2 `Red`

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

Error messages:

1. `Not reducible`

7.5.3 `Hnf`

This tactic applies to any goal. It replaces the current goal with its head normal form according to the $\beta\delta\iota$-reduction rules. `Hnf` does not produce a real head normal form but either a product or an applicative term in head normal form or a variable.

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

Remark: The δ rule will only be applied to transparent constants (i.e. which have not been frozen with an `Opaque` command; see section 5.2.4).

7.5.4 `Simpl`

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

7.5.5 `Unfold` *qualid*

This tactic applies to any goal. The argument *qualid* must denote a defined transparent constant or local definition (see section 1.3.2 and 5.2.5). The tactic `Unfold` applies the δ rule to each occurrence of the constant to which *qualid* refers in the current goal and then replaces it with its $\beta\iota$-normal form.

Error messages:

1. *qualid* `does not denote an evaluable constant` is printed.

Variants:

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

2. `Unfold` num_1^1 `...` num_i^1 $qualid_1$ `...` num_1^n `...` num_j^n $qualid_n$
   The lists $num_1^1, \ldots, num_i^1$ and $num_1^n, \ldots, num_j^n$ are to specify the occurrences of $qualid_1$, $\ldots$, $qualid_n$ to be unfolded. Occurrences are located from left to right in the linear notation of terms.
   Error message: `bad occurrence numbers of` $qualid_i$

7.5.6 `Fold` ***term***

This tactic applies to any goal. *term* is reduced using the `Red` tactic. Every occurrence of the resulting term in the goal is then substituted for *term*.

Variants:

1. `Fold` $term_1 \ldots term_n$
   Equivalent to `Fold` $term_1$`;`...`;` `Fold` $term_n$.

7.5.7 `Pattern` *term*

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

1. replacing all occurrences of *term* in `T` with a fresh variable
2. abstracting this variable
3. applying the abstracted goal to *term*

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

Variants:

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

2. `Pattern` num_1^1 `...` $num_{n_1}^1$ $term_1$ `...`num_1^m `...` $num_{n_m}^m$ $term_m$
   Will process occurrences $num_1^1, \ldots, num_i^1$ of $term_1, \ldots, num_1^m, \ldots, num_j^m$ of $term_m$ starting from $term_m$. Starting from a goal `(P t`$_1$`... t`$_m$`)` with the `t`$_i$ which do not occur in P, the tactic `Pattern t`$_1$`... t`$_m$ generates the equivalent goal `([x`$_1$`:A`$_1$`]... [x`$_m$`:A`$_m$`](P 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.

7.5.8 Conversion tactics applied to hypotheses

conv_tactic `in` $ident_1 \ldots ident_n$
Applies the conversion tactic *conv_tactic* to the hypotheses $ident_1, \ldots, ident_n$. The tactic *conv_tactic* is any of the conversion tactics listed in this section.

Error messages:

1. `No such hypothesis:` *ident*.

7.6 Introductions

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

7.6.1 `Constructor` *num*

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

Error messages:

1. `Not an inductive product`
2. `Not enough Constructors`

Variants:

1. `Constructor`
   This tries `Constructor 1` then `Constructor 2,...`, then `Constructor` *n* where *n* if the number of constructors of the head of the goal.

2. `Constructor` *num* `with` *bindings_list*
   Let `ci` be the `i`-th constructor of `I`, then `Constructor i with` *bindings_list* is equivalent to `Intros; Apply ci with` *bindings_list*.

   Warning: the terms in the *bindings_list* are checked in the context where `Constructor` is executed and not in the context where `Apply` is executed (the introductions are not taken into account).

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

4. `Exists` *bindings_list*
   Applies if `I` has only one constructor, for instance in the case of existential quantification $\exists x \cdot P(x)$. It is equivalent to `Intros; Constructor 1 with` *bindings_list*.

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

6. `Left` *bindings_list*, `Right` *bindings_list*, `Split` *bindings_list*
   Are equivalent to the corresponding `Constructor` i `with` *bindings_list*.

7.7 Eliminations (Induction and Case Analysis)

Elimination tactics are useful to prove statements by induction or case analysis. Indeed, they make use of the elimination (or induction) principles generated with inductive definitions (see section 4.5).

7.7.1 `NewInduction` *term*

This tactic applies to any goal. The type of the argument *term* must be an inductive constant. Then, the tactic `NewInduction` generates subgoals, one for each possible form of *term*, i.e. one for each constructor of the inductive type.

The tactic `NewInduction` automatically replaces every occurrences of *term* in the conclusion and the hypotheses of the goal. It automatically adds induction hypotheses (using names of the form `IHn1`) to the local context. If some hypothesis must not be taken into account in the induction hypothesis, then it needs to be removed first (you can also use the tactic `Elim`, see below).

`NewInduction` works also when *term* is an identifier denoting a quantified variable of the conclusion of the goal. Then it behaves as `Intros until` *ident*`; NewInduction` *ident*.

```
Coq < Lemma induction_test : (n:nat) n=n -> (le n n).
1 subgoal

  ============================
   (n:nat)n=n->(le n n)

Coq < Intros n H.
1 subgoal

  n : nat
  H : n=n
  ============================
   (le n n)

Coq < NewInduction n.
2 subgoals

  H : O=O
  ============================
   (le O O)
subgoal 2 is:
 (le (S n) (S n))
```

Error messages:

1. `Not an inductive product`

2. `Cannot refine to conclusions with meta-variables`
   As `NewInduction` uses `Apply`, see section 7.3.6 and the variant `Elim ... with ...` below.

Variants:

1. `NewInduction` *num* is analogous to `NewInduction` *ident* (when *ident* a quantified variable of the goal) but for the *num*-th non-dependent premise of the goal.

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

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

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

5. `Elim` *term* `with` ref_1 `:=` $term_1$ `...` ref_n `:=` $term_n$
   Provides also `Elim` with values for instantiating premises by associating explicitly variables (or non dependent products) with their intended instance.

6. `Elim` $term_1$ `using` $term_2$
   Allows the user to give explicitly an elimination predicate $term_2$ which is not the standard one for the underlying inductive type of $term_1$. Each of the $term_1$ and $term_2$ is either a simple term or a term with a bindings list (see 7.3.11).

7. `ElimType` *form*
   The argument *form* must be inductively defined. `ElimType I` is equivalent to `Cut I. Intro H`*n*`; Elim H`*n*`; Clear H`*n* Therefore the hypothesis `H`*n* will not appear in the context(s) of the subgoal(s).
   Conversely, if `t` is a term of (inductive) type `I` and which does not occur in the goal then `Elim t` is equivalent to `ElimType I; 2: Exact t.`

   Error message: `Impossible to unify ... with ...`
   Arises when *form* needs to be applied to parameters.

8. `Induction` *ident*
   This is a deprecated tactic, which behaves as `Intros until` *ident*`; Elim` *ident* when *ident* is a quantified variable of the goal, and similarly as `NewInduction` *ident*, when *ident* is an hypothesis (except in the way induction hypotheses are named).

9. `Induction` *num*
   This is a deprecated tactic, which 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.

7.7.2 `NewDestruct` *term*

The tactic `NewDestruct` is used to perform case analysis without recursion. Its behaviour is similar to `NewInduction` *term* except that no induction hypotheses is generated. It applies to any goal and the type of *term* must be inductively defined. `NewDestruct` works also when *term* is an identifier denoting a quantified variable of the conclusion of the goal. Then it behaves as `Intros until` *ident*`; NewDestruct` *ident*.

Variants:

1. `NewDestruct` *num*
   Is analogous to `NewDestruct` *ident* (when *ident* a quantified variable of the goal), but for the *num*-th non-dependent premise of the goal.

2. `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.

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

4. `Destruct` *ident*
   This is a deprecated tactic, which behaves as `Intros until` *ident*`; Case` *ident* when *ident* is a quantified variable of the goal.

5. `Destruct` *num*
   This is a deprecated tactic, which 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.

7.7.3 `Intros` *pattern*

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

A pattern is either:

- the wildcard: _
- a variable
- a list of patterns: $p_1\ \ldots\ p_n$

- a disjunction of patterns: `[`p_1 `|` ... `|` p_n `]`
- a conjunction of patterns: `(` p_1 `,` ... `,` p_n `)`

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

- introduction on the wildcard do the introduction and then immediately clear (cf 7.3.2) the corresponding hypothesis;
- introduction on a variable behaves like described in 7.3.5;
- introduction over a list of patterns p_1 ... p_n is equivalent to the sequence of introductions over the patterns namely: `Intros` p_1`;...;` `Intros` p_n, the goal should start with at least n products;
- introduction over a disjunction of patterns $[p_1 \mid \ \ldots \mid p_n]$, it introduces a new variable X, its type should be an inductive definition with n constructors, then it performs a case analysis over X (which generates n subgoals), it clears X and performs on each generated subgoals the corresponding `Intros` p_i tactic;
- introduction over a conjunction of patterns $(p_1, \ldots, p_n)$, it introduces a new variable X, its type should be an inductive definition with 1 constructor with (at least) n arguments, then it performs a case analysis over X (which generates 1 subgoal with at least n products), it clears X and performs an introduction over the list of patterns p_1 ... p_n.

```
Coq < Lemma intros_test : (A,B,C:Prop)(A\/(B/\C))->(A->C)->C.
1 subgoal

  ============================
   (A,B,C:Prop)A\/B/\C->(A->C)->C

Coq < Intros A B C [a|(_,c)] f.
2 subgoals

  A : Prop
  B : Prop
  C : Prop
  a : A
  f : A->C
  ============================
   C
subgoal 2 is:
 C

Coq < Apply (f a).
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  c : C
  f : A->C
  ============================
```

```
  C

Coq < Proof c.
intros_test is defined
```

7.7.4 `Double Induction` *num*$_1$ *num*$_2$

This tactic applies to any goal. If the *num*$_1$th and *num*$_2$th premises of the goal have an inductive type, then this tactic performs double induction on these premises. For instance, if the current goal is `(n,m:nat)(P n m)` then, `Double Induction 1 2` yields the four cases with their respective inductive hypothesis. In particular the case for `(P (S n) (S m))` with the inductive hypothesis about both `n` and `m`.

7.7.5 `Decompose [` *ident*$_1$ `...` *ident*$_n$ `]` *term*

This tactic allows to recursively decompose a complex proposition in order to obtain atomic ones. Example:

```
Coq < Lemma ex1: (A,B,C:Prop)(A/\B/\C \/ B/\C \/ C/\A) -> C.
1 subgoal

  ============================
   (A,B,C:Prop)A/\B/\C\/B/\C\/C/\A->C

Coq < Intros A B C H; Decompose [and or] H; Assumption.
Subtree proved!

Coq < Qed.
```

`Decompose` does not work on right-hand sides of implications or products.

Variants:

1. `Decompose Sum` *term* This decomposes sum types (like `or`).
2. `Decompose Record` *term* This decomposes record types (inductive types with one constructor, like `and` and `exists` and those defined with the `Record` macro, see p. 43).

7.8 Equality

These tactics use the equality `eq:(A:Set)A->A->Prop` defined in file `Logic.v` and the equality `eqT:(A:Type)A->A->Prop` defined in file `Logic_Type.v` (see section 3.1.1). They are simply written `t=u` and `t==u`, respectively. In the following, the notation `t=u` will represent either one of these two equalities.

7.8.1 `Rewrite` *term*

This tactic applies to any goal. The type of *term* must have the form
   `(`x_1`:`A_1`) ... (`x_n`:`A_n`)` *term*$_1$=*term*$_2$.
Then `Rewrite` *term* replaces every occurrence of *term*$_1$ by *term*$_2$ in the goal. Some of the variables x_1 are solved by unification, and some of the types $A_1, \ldots, A_n$ become new subgoals.

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

Error messages:

1. `The term provided does not end with an equation`
2. `Tactic generated a subgoal identical to the original goal`
   This happens if $term_1$ does not occur in the goal.

Variants:

1. `Rewrite ->` *term*
   Is equivalent to `Rewrite` *term*
2. `Rewrite <-` *term*
   Uses the equality $term_1$=$term_2$ from right to left
3. `Rewrite` *term* `in` *ident*
   Analogous to `Rewrite` *term* but rewriting is done in the hypothesis named *ident*.
4. `Rewrite ->` *term* `in` *ident*
   Behaves as `Rewrite` *term* `in` *ident*.
5. `Rewrite <-` *term* `in` *ident*
   Uses the equality $term_1$=$term_2$ from right to left to rewrite in the hypothesis named *ident*.

7.8.2 `CutRewrite ->` $term_1$ `=` $term_2$

This tactic acts like `Replace` $term_1$ `with` $term_2$ (see below).

7.8.3 `Replace` $term_1$ `with` $term_2$

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

7.8.4 `Reflexivity`

This tactic applies to a goal which has the form `t=u`. It checks that `t` and `u` are convertible and then solves the goal. It is equivalent to `Apply refl_equal` (or `Apply refl_equalT` for an equality in the Type universe).

Error messages:

1. `The conclusion is not a substitutive equation`
2. `Impossible to unify ... with ..`

7.8.5 `Symmetry`

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

7.8.6 `Transitivity` *term*

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

7.9 Equality and inductive sets

We describe in this section some special purpose tactics dealing with equality and inductive sets or types. These tactics use the equalities `eq:(A:Set)A->A->Prop` defined in file `Logic.v` and `eqT:(A:Type)A->A->Prop` defined in file `Logic_Type.v` (see section 3.1.1). They are written `t=u` and `t==u`, respectively. In the following, unless it is stated otherwise, the notation `t=u` will represent either one of these two equalities.

7.9.1 `Decide Equality`

This tactic solves a goal of the form $(x, y : R)\{x = y\} + \{\sim x = y\}$, where R is an inductive type such that its constructors do not take proofs or functions as arguments, nor objects in dependent types.

Variants:

1. `Decide Equality` $term_1$ $term_2$ `.`
   Solves a goal of the form `{`$term_1$`=`$term_2$`}+{~`$term_1$`=`$term_2$`}`.

7.9.2 `Compare` $term_1$ $term_2$

This tactic compares two given objects $term_1$ and $term_2$ of an inductive datatype. If G is the current goal, it leaves the sub-goals $term_1$=$term_2$ `->` G and ~$term_1$=$term_2$ `->` G. The type of $term_1$ and $term_2$ must satisfy the same restrictions as in the tactic `Decide Equality`.

7.9.3 `Discriminate` *ident*

This tactic proves any goal from an absurd hypothesis stating that two structurally different terms of an inductive set are equal. For example, from the hypothesis `(S (S O))=(S O)` we can derive by absurdity any proposition. Let *ident* be a hypothesis of type $term_1$ = $term_2$ in the local context, $term_1$ and $term_2$ being elements of an inductive set. To build the proof, the tactic traverses the normal forms[4] of $term_1$ and $term_2$ looking for a couple of subterms `u` and `w` (`u` subterm of the normal form of $term_1$ and `w` subterm of the normal form of $term_2$), placed at the same positions and whose head symbols are two different constructors. If such a couple of subterms exists, then the proof of the current goal is completed, otherwise the tactic fails.

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

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

Error messages:

1. *ident* `Not a discriminable equality`
   occurs when the type of the specified hypothesis is not an equation.

Variants:

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

2. `Discriminate`
   It applies to a goal of the form ~*term*$_1$=*term*$_2$ and it is equivalent to: `Unfold not; Intro` *ident* `;Discriminate` *ident*.

   Error messages:

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

7.9.4 `Injection` *ident*

The `Injection` tactic is based on the fact that constructors of inductive sets are injections. That means that if c is a constructor of an inductive set, and if $(c\ \vec{t_1})$ and $(c\ \vec{t_2})$ are two terms that are equal then $\vec{t_1}$ and $\vec{t_2}$ are equal too.

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

- $\mathtt{n}_1$ and $\mathtt{n}_2$ should not be syntactically equal,
- there must not exist any couple of subterms `u` and `w`, `u` subterm of $\mathtt{n}_1$ and `w` subterm of $\mathtt{n}_2$, placed in the same positions and having different constructors as head symbols.

If these conditions are satisfied, then, the tactic derives the equality of all the subterms of *term*$_1$ and *term*$_2$ placed in the same positions and puts them as antecedents of the current goal.

Example: Consider the following goal:

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

Coq < Variable  P : list -> Prop.
```

```
Coq < Show.
1 subgoal

  l : list
  n : nat
  H : (P nil)
  H0 : (cons n l)=(cons O nil)
  ============================
   (P l)

Coq < Injection H0.
1 subgoal

  l : list
  n : nat
  H : (P nil)
  H0 : (cons n l)=(cons O nil)
  ============================
   l=nil->n=O->(P l)
```

Beware that `Injection` yields always an equality in a sigma type whenever the injected object has a dependent type.

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

Error messages:

1. *ident* `is not a projectable equality` occurs when the type of the hypothesis id does not verify the preconditions.

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

Variants:

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

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

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

7.9.5 `Simplify_eq` *ident*

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

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

Variants:

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

2. `Simplify_eq` If the current goal has form $\sim t_1 = t_2$, then this tactic does `Hnf; Intro` *ident*`;` `Simplify_eq` *ident*.

7.9.6 `Dependent Rewrite ->` *ident*

This tactic applies to any goal. If *ident* has type `(existS A B a b)=(existS A B a' b')` in the local context (i.e. each term of the equality has a sigma type $\{a : A \& (B\ a)\}$) this tactic rewrites `a` into `a'` and `b` into `b'` in the current goal. This tactic works even if B is also a sigma type. This kind of equalities between dependent pairs may be derived by the injection and inversion tactics.

Variants:

1. `Dependent Rewrite <-` *ident*
   Analogous to `Dependent Rewrite ->` but uses the equality from right to left.

7.10 Inversion

7.10.1 `Inversion` *ident*

Let the type of *ident* in the local context be $(I\ \vec{t})$, where I is a (co)inductive predicate. Then, `Inversion` applied to *ident* derives for each possible constructor c_i of $(I\ \vec{t})$, **all** the necessary conditions that should hold for the instance $(I\ \vec{t})$ to be proved by c_i.

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

Variants:

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

2. `Inversion_clear` *ident*
   That does `Inversion` and then erases *ident* from the context.

3. `Inversion` *ident* `in` *ident*$_1$ ... *ident*$_n$
   Let *ident*$_1$ ... *ident*$_n$, be identifiers in the local context. This tactic behaves as generalizing *ident*$_1$ ... *ident*$_n$, and then performing `Inversion`.

4. `Inversion_clear` *ident* `in` *ident*$_1$ ... *ident*$_n$
   Let *ident*$_1$ ... *ident*$_n$, be identifiers in the local context. This tactic behaves as generalizing *ident*$_1$ ... *ident*$_n$, and then performing `Inversion_clear`.

5. `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.

6. `Dependent Inversion_clear` *ident*
   Like `Dependant Inversion`, except that *ident* is cleared from the local context.

7. `Dependent Inversion` *ident* `with` *term*
   This variant allow to give the good generalization of the goal. It is useful when the system fails to generalize the goal automatically. If *ident* has type $(I\ \vec{t})$ and I has type $(\vec{x} : \vec{T})s$, then *term* must be of type $I : (\vec{x} : \vec{T})(I\ \vec{x}) \to s'$ where s' is the type of the goal.

8. `Dependent Inversion_clear` *ident* `with` *term*
   Like `Dependant Inversion ... with` but clears *ident*from the local context.

9. `Inversion` *ident* `using` *ident′*
   Let *ident* have type $(I\ \vec{t})$ (I an inductive predicate) in the local context, and *ident′* be a (dependent) inversion lemma. Then, this tactic refines the current goal with the specified lemma.

10. `Inversion` *ident* `using` *ident′* `in` $ident_1 \ldots ident_n$
    This tactic behaves as generalizing $ident_1 \ldots ident_n$, then doing `Inversion`*ident* `using` *ident′*.

11. `Simple Inversion` *ident*
    It is a very primitive inversion tactic that derives all the necessary equalities but it does not simplify the constraints as `Inversion` do.

See also: 8.4 for detailed examples

7.10.2 `Derive Inversion` *ident* `with` $(\vec{x} : \vec{T})(I\ \ \vec{t})$ `Sort` *sort*

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

Variants:

1. `Derive Inversion_clear` *ident* `with` $(\vec{x} : \vec{T})(I\ \vec{t})$ `Sort` *sort*
   When applied it is equivalent to having inverted the instance with the tactic `Inversion` replaced by the tactic `Inversion_clear`.

2. `Derive Dependent Inversion` *ident* `with` $(\vec{x} : \vec{T})(I\ \vec{t})$ `Sort` *sort*
   When applied it is equivalent to having inverted the instance with the tactic `Dependent Inversion`.

3. `Derive Dependent Inversion_clear` *ident* `with` $(\vec{x} : \vec{T})(I\ \vec{t})$ `Sort` *sort*
   When applied it is equivalent to having inverted the instance with the tactic `Dependent Inversion_clear`.

See also: 8.4 for examples

7.10.3 `Quote` *ident*

-level approach

This kind of inversion has nothing to do with the tactic `Inversion` above. This tactic does `Change (`*ident*` t)`, where `t` is a term build in order to ensure the convertibility. In other words, it does inversion of the function *ident*. This function must be a fixpoint on a simple recursive datatype: see 8.6 for the full details.

Error messages:

1. `Quote: not a simple fixpoint`
   Happens when `Quote` is not able to perform inversion properly.

Variants:

1. `Quote` *ident* `[` $ident_1$ `...`$ident_n$ `]`
   All terms that are build only with $ident_1 \dots ident_n$ will be considered by `Quote` as constants rather than variables.

See also: file `theories/DEMOS/DemoQuote.v` in the distribution

7.11 Automatizing

7.11.1 `Auto`

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

By default, Auto only uses the hypotheses of the current goal and the hints of the database named "core".

Variants:

1. `Auto` *num*
   Forces the search depth to be *num*. The maximal search depth is 5 by default.

2. `Auto with` $ident_1$ `...` $ident_n$
   Uses the hint databases $ident_1 \dots ident_n$ in addition to the database "core". See section 7.12 for the list of pre-defined databases and the way to create or extend a database. This option can be combined with the previous one.

3. `Auto with *`
   Uses all existing hint databases, minus the special database "v62". See section 7.12

4. `Trivial`
   This tactic is a restriction of `Auto` that is not recursive and tries only hints which cost is 0. Typically it solves trivial equalities like $X = X$.

5. `Trivial with` $ident_1$ `...` $ident_n$

6. `Trivial with *`

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

See also: section 7.12

7.11.2 `EAuto`

This tactic generalizes `Auto`. In contrast with the latter, `EAuto` uses unification of the goal against the hints rather than pattern-matching (in other words, it uses `EApply` instead of `Apply`). As a consequence, `EAuto` can solve such a goal:

```
Coq < Hints Resolve ex_intro.
Warning: the hint: EApply ex_intro will only be used by EAuto

Coq < Goal (P:nat->Prop)(P O)->(EX n | (P n)).
1 subgoal

  ============================
   (P:(nat->Prop))(P O)->(EX n:nat | (P n))

Coq < EAuto.
Subtree proved!
```

Note that `ex_intro` should be declared as an hint.

See also: section 7.12

7.11.3 `Prolog [` *term*$_1$ `...` *term*$_n$ `]` *num*

This tactic, implemented by Chet Murthy, is based upon the concept of existential variables of Gilles Dowek, stating that resolution is a kind of unification. It tries to solve the current goal using the `Assumption` tactic, the `Intro` tactic, and applying hypotheses of the local context and terms of the given list `[` *term*$_1$ `...` *term*$_n$ `]`. It is more powerful than `Auto` since it may apply to any theorem, even those of the form `(x:A)(P x) -> Q` where `x` does not appear free in `Q`. The maximal search depth is *num*.

Error messages:

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

7.11.4 `Tauto`

This tactic implements a decision procedure for intuitionistic propositional calculus based on the contraction-free sequent calculi LJT* of Roy Dyckhoff [44]. Note that `Tauto` succeeds on any instance of an intuitionistic tautological proposition. `Tauto` unfolds negations and logical equivalence but does not unfold any other definition.

The following goal can be proved by `Tauto` whereas `Auto` would fail:

```
Coq <   Goal (x:nat)(P:nat->Prop)x=O\/(P x)->~x=O->(P x).
1 subgoal

  ============================
   (x:nat; P:(nat->Prop))x=O\/(P x)->~x=O->(P x)

Coq <   Intros.
1 subgoal

  x : nat
  P0 : nat->Prop
  H : x=O\/(P0 x)
  H0 : ~x=O
  ============================
   (P0 x)

Coq <   Tauto. (* Auto would fail *)
Subtree proved!
```

Moreover, if it has nothing else to do, `Tauto` performs introductions. Therefore, the use of `Intros` in the previous proof is unnecessary. `Tauto` can for instance prove the following:

```
Coq <   Goal (A:Prop)(P:nat->Prop)(A \/ (x:nat) ~A -> (P x)) -> (x:nat) ~A -> (P x).
1 subgoal

  ============================
   (A:Prop; P:(nat->Prop))A\/((x:nat)~A->(P x))->(x:nat)~A->(P x)

Coq <   Tauto.
Subtree proved!
```

Remark: In contrast, `Tauto` cannot solve the following goal

```
Coq < Goal (A:Prop)(P:nat->Prop)(A \/ (x:nat) ~A -> (P x)) -> (x:nat) ~~ (A \/ (P x)).
```

because `(x:nat) ~A -> (P x)` cannot be treated as atomic and an instantiation of `x` is necessary.

7.11.5 `Intuition` *tactic*

The tactic `Intuition` takes advantage of the search-tree builded by the decision procedure involved in the tactic `Tauto`. It uses this information to generate a set of subgoals equivalent to the original one (but simpler than it) and applies the tactic *tactic* to them [81]. If this tactic fails on some goals then `Intuition` fails. In fact, `Tauto` is simply `Intuition Failtac`.

For instance, the tactic `Intuition Auto` applied to the goal

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

internally replaces it by the equivalent one:

```
(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 reenginered by David Delahaye using mainly the tactic language (see chapter 10). The code is now quite shorter and a significant increase in performances has been noticed. The general behavior with respect to dependent types, unfolding and introductions has slightly changed to get clearer semantics. This may lead to some incompatibilities.

Variants:

1. `Intuition`
   Is equivalent to `Intuition Auto with *`.

See also: file `contrib/Rocq/DEMOS/Demo_tauto.v`

7.11.6 `Omega`

The tactic `Omega`, due to Pierre Crégut, is an automatic decision procedure for Prestburger arithmetic. It solves quantifier-free formulae build with `~`, `\/`, `/\`, `->` on top of equations and inequations on both the type `nat` of natural numbers and `Z` of binary integers. This tactic must be loaded by the command `Require Omega`. See the additional documentation about `Omega` (chapter 15).

7.11.7 `Ring` *term*$_1$ ... *term*$_n$

This tactic, written by Samuel Boutin and Patrick Loiseleur, does AC rewriting on every ring. The tactic must be loaded by `Require Ring` under `coqtop` or `coqtop -full`. The ring must be declared in the `Add Ring` command (see 18). The ring of booleans is predefined; if one wants to use the tactic on `nat` one must do `Require ArithRing`; for `Z`, do `Require ZArithRing`.

term$_1$, ..., *term*$_n$ must be subterms of the goal conclusion. `Ring` normalize these terms w.r.t. associativity and commutativity and replace them by their normal form.

Variants:

1. `Ring` When the goal is an equality $t_1 = t_2$, it acts like `Ring` t_1 t_2 and then simplifies or solves the equality.

2. `NatRing` is a tactic macro for `Repeat Rewrite S_to_plus_one; Ring`. The theorem `S_to_plus_one` is a proof that `(n:nat)(S n)=(plus (S O) n)`.

Example:

```
Coq < Require ZArithRing.

Coq < Goal (a,b,c:Z)`(a+b+c)*(a+b+c)
Coq <               = a*a + b*b + c*c + 2*a*b + 2*a*c + 2*b*c`.

Coq < Intros; Ring.
Subtree proved!
```

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

See also: Chapter 18 for more detailed explanations about this tactic.

7.11.8 `Field`

This tactic written by David Delahaye and Micaela Mayero solves equalities using commutative field theory. Denominators have to be non equal to zero and, as this is not decidable in general, this tactic may generate side conditions requiring some expressions to be non equal to zero. This tactic must be loaded by `Require Field`. Field theories are declared (as for `Ring`) with the `Add Field` command.

Example:

```
Coq < Require Reals.

Coq < Goal (x,y:R)``x*y>0`` -> ``x*((1/x)+x/(x+y)) == -(1/y)*y*(-(x*x/(x+y))-1)``.

Coq < Intros; Field.
1 subgoal

  x : R
  y : R
  H : ``x*y > 0``
  ============================
   ``x*((x+y)*y) <> 0``
```

7.11.9 `Add Field`

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

`Add Field` *A Aplus Amult Aone Azero Aopp Aeq Ainv Rth Tinvl*

where *A* is a term of type `Type`, *Aplus* is a term of type `A->A->A`, *Amult* is a term of type `A->A->A`, *Aone* is a term of type `A`, *Azero* is a term of type `A`, *Aopp* is a term of type `A->A`, *Aeq* is a term of type `A->bool`, *Ainv* is a term of type `A->A`, *Rth* is a term of type `(Ring_Theory` *A Aplus Amult Aone Azero Ainv Aeq*`)`, and *Tinvl* is a term of type `(n:`*A*`)~(n==`*Azero*`)->(`*Amult* `(`*Ainv* `n) n)==`*Aone*. To build a ring theory, refer to chapter 18 for more details.

This command adds also an entry in the ring theory table if this theory is not already declared. So, it is useless to keep, for a given type, the `Add Ring` command if you declare a theory with `Add Field`, except if you plan to use specific features of `Ring` (see chapter 18). However, the module `Ring` is not loaded by `Add Field` and you have to make a `Require Ring` if you want to call the `Ring` tactic.

Variants:

1. `Add Field` *A Aplus Amult Aone Azero Aopp Aeq Ainv Rth Tinvl*
   `with minus:=`*Aminus*
   Adds also the term *Aminus* which must be a constant expressed by means of *Aopp*.

2. `Add Field` *A Aplus Amult Aone Azero Aopp Aeq Ainv Rth Tinvl*
   `with div:=`*Adiv*
   Adds also the term *Adiv* which must be a constant expressed by means of *Ainv*.

See also: file `theories/Reals/Rbase.v` for an example of instantiation,
theory `theories/Reals` for many examples of use of `Field`.

See also: [32] for more details regarding the implementation of `Field`.

7.11.10 `Fourier`

This tactic written by Loïc Pottier solves linear inequations on real numbers using Fourier's method ([53]). This tactic must be loaded by `Require Fourier`.

Example:

```
Coq < Require Reals.

Coq < Require Fourier.

Coq < Goal (x,y:R)``x < y``->``y+1 >= x-1``.

Coq < Intros; Fourier.
Subtree proved!
```

7.11.11 `AutoRewrite [` $ident_1$ ... $ident_n$ `]`

This tactic [5] carries out rewritings according the rewriting rule bases $ident_1$ ... $ident_n$.

Each rewriting rule of a base $ident_i$ is applied to the main subgoal until it fails. Once all the rules have been processed, if the main subgoal has progressed (e.g., if it is distinct from the initial main goal) then the rules of this base are processed again. If the main subgoal has not progressed then the next base is processed. For the bases, the behavior is exactly similar to the processing of the rewriting rules.

The rewriting rule bases are built with the `Hint Rewrite` vernacular command.

Warning: This tactic may loop if you build non terminating rewriting systems.

Variant:

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

7.11.12 `Hint Rewrite [` $term_1$ ... $term_n$ `] in` *ident*

This vernacular command adds the terms $term_1$ ... $term_n$ (their types must be equalities) in the rewriting base *ident* with the default orientation (left to right).

This command is synchronous with the section mechanism (see 2.4): when closing a section, all aliases created by `Hint Rewrite` in that section are lost. Conversely, when loading a module, all `Hint Rewrite` declarations at the global level of that module are loaded.

Variants:

1. `Hint Rewrite -> [` $term_1$ ...$term_n$ `] in` *ident*
   This is strictly equivalent to the command above (we only precise the orientation which is the default one).

2. `Hint Rewrite <- [` $term_1$ ...$term_n$ `] in` *ident*
   Adds the rewriting rules $term_1$ ... $term_n$ with a right-to-left orientation in the base *ident*.

[5] The behavior of this tactic has much changed compared to the versions available in the previous distributions (V6). This may cause significant changes in your theories to obtain the same result. As a drawback of the reenginering of the code, this tactic has also been completely revised to get a very compact and readable version.

3. `Hint Rewrite [` *term*$_1$ `...`*term*$_n$ `] in` *ident* `using` *tactic*
   When the rewriting rules *term*$_1$ ... *term*$_n$ in *ident* will be used, the tactic *tactic* will be applied to the generated subgoals, the main subgoal excluded.

See also: 8.5 for examples showing the use of this tactic.

See also: file `contrib/Rocq/DEMOS/Demo_AutoRewrite.v`

7.12 The hints databases for Auto and EAuto

The hints for Auto and EAuto have been reorganized since Coq 6.2.3. They are stored in several databases. Each databases maps head symbols to list of hints. One can use the command `Print Hint` *ident* to display the hints associated to the head symbol *ident* (see 7.12.2). Each hint has a name, a cost that is an nonnegative integer, and a pattern. The hint is tried by `Auto` if the conclusion of current goal matches its pattern, and after hints with a lower cost. The general command to add a hint to a database is:

`Hint` *name* `:` *database* `:=` *hint_definition*

where *hint_definition* is one of the following expressions:

- `Resolve` *term*
  This command adds `Apply` *term* to the hint list with the head symbol of the type of *term*. The cost of that hint is the number of subgoals generated by `Apply` *term*.

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

  If the inferred type of *term* does contain a dependent quantification on a predicate, it is added to the hint list of `EApply` instead of the hint list of `Apply`. In this case, a warning is printed since the hint is only used by the tactic `EAuto` (see 7.11.2). A typical example of hint that is used only by `EAuto` is a transitivity lemma.

  Error messages:

  1. `Bound head variable`
     The head symbol of the type of *term* is a bound variable such that this tactic cannot be associated to a constant.
  2. *term* `cannot be used as a hint`
     The type of *term* contains products over variables which do not appear in the conclusion. A typical example is a transitivity axiom. In that case the `Apply` tactic fails, and thus is useless.

- `Immediate` *term*

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

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

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

Error messages:

1. `Bound head variable`

2. *term* `cannot be used as a hint`

- `Constructors` *ident*

  If *ident* is an inductive type, this command adds all its constructors as hints of type `Resolve`. Then, when the conclusion of current goal has the form `(`*ident* `...)`, `Auto` will try to apply each constructor.

  Error messages:

  1. *ident* `is not an inductive type`
  2. *ident* `not declared`

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

- `Extern` *num pattern tactic*
  This hint type is to extend Auto with tactics other than `Apply` and `Unfold`. For that, we must specify a cost, a pattern and a tactic to execute. Here is an example:

  ```
  Hint discr : core := Extern 4 ~(?=?) Discriminate.
  ```

  Now, when the head of the goal is a disequality, `Auto` will try `Discriminate` if it does not succeed to solve the goal with hints with a cost less than 4.

  One can even use some sub-patterns of the pattern in the tactic script. A sub-pattern is a question mark followed by a number like `?1` or `?2`. Here is an example:

```
Coq < Require EqDecide.

Coq < Require PolyList.

Coq < Hint eqdec1 : eqdec := Extern 5 {?1=?2}+{~ (?1=?2)}
Coq <                                 Generalize ?1 ?2; Decide Equality.

Coq <
Coq < Goal (a,b:(list nat*nat)){a=b}+{~a=b}.
1 subgoal
```

```
  ============================
    (a,b:(list nat*nat)){a=b}+{~a=b}

Coq < Info Auto with eqdec.
 == Intro a; Intro b; Generalize a b; Decide Equality; Generalize a0 p;
       Decide Equality.
    Generalize b0 n0; Decide Equality.

    Generalize a1 n; Decide Equality.

Subtree proved!
```

Remark: There is currently (in the 7.3 release) no way to do pattern-matching on hypotheses.

Variants:

1. `Hint` *ident* `:` $ident_1$ `...` $ident_n$ `:=` *hint_expression*
   This syntax allows to put the same hint in several databases.

   Remark: The current implementation of `Auto` has no optimization about hint duplication: if the same hint is present in two databases given as arguments to `Auto`, it will be tried twice. We recommend to put the same hint in two different databases only if you never use those databases together.

2. `Hint` *ident* `:=` *hint_expression*
   If no database name is given, the hint is registered in the "core" database.

   Remark: We do not recommend to put hints in this database in your developpements, except when the `Hint` command is inside a section. In this case the hint will be thrown when closing the section (see 7.12.3)

There are shortcuts that allow to define several goal at once:

- `Hints Resolve` $ident_1$ `...` $ident_n$ `:` *ident* `.`
  This command is a shortcut for the following ones:

  `Hint` $ident_1$ `:` *ident* `:= Resolve` $ident_1$
  `...`
  `Hint` $ident_1$ `:` *ident*`:= Resolve` $ident_1$

  Notice that the hint name is the same that the theorem given as hint.

- `Hints Immediate` $ident_1$ `...` $ident_n$ `:` *ident* `.`

- `Hints Unfold` $qualid_1$ `...` $qualid_n$ `:` *ident* `.`

7.12.1 Hint databases defined in the Coq standard library

Several hint databases are defined in the Coq standard library. There is no systematic relation between the directories of the library and the databases.

core This special database is automatically used by `Auto`. It contains only basic lemmas about negation, conjunction, and so on from. Most of the hints in this database come from the `INIT` and `LOGIC` directories.

arith This databases contains all lemmas about Peano's arithmetic proven in the directories `INIT` and `ARITH`

zarith contains lemmas about binary signed integers from the directories `theories/ZARITH` and `tactics/contrib/Omega`. It contains also a hint with a high cost that calls Omega.

bool contains lemmas about booleans, mostly from directory `theories/BOOL`.

datatypes is for lemmas about about lists, trees, streams and so on that are proven in `LISTS`, `TREES` subdirectories.

sets contains lemmas about sets and relations from the directory `SETS` and `RELATIONS`.

There is also a special database called "v62". It contains all things that are currently hinted in the 6.2.x releases. It will not be extended later. It is not included in the hint databases list used in the "Auto with *" tactic.

The only purpose of the database "v62" is to ensure compatibility for old developpements with further versions of Coq. If you have a developpement that used to compile with 6.2.2 and that not compiles with 6.2.4, try to replace "Auto" with "Auto with v62" using the script documented below. This will ensure your developpement will compile will further releases of Coq.

To write a new developpement, or to update a developpement not finished yet, you are strongly advised NOT to use this database, but the pre-defined databases. Furthermore, you are advised not to put your own Hints in the "core" database, but use one or several databases specific to your developpement.

7.12.2 `Print Hint`

This command displays all hints that apply to the current goal. It fails if no proof is being edited, while the two variants can be used at every moment.

Variants:

1. `Print Hint` *ident*
   This command displays only tactics associated with *ident* in the hints list. This is independent of the goal being edited, to this command will not fail if no goal is being edited.

2. `Print Hint *`
   This command displays all declared hints.

3. `Print HintDb` *ident*
   This command displays all hints from database *ident*.

7.12.3 Hints and sections

Like grammar rules and structures for the `Ring` tactic, things added by the `Hint` command will be erased when closing a section.

Conversely, when the user does `Require A.`, all hints of the module `A` that are not defined inside a section are loaded.

7.13 Tacticals

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

7.13.1 `Idtac`

The constant `Idtac` is the identity tactic: it leaves any goal unchanged.

7.13.2 `Fail`

The tactic `Fail` is the always-failing tactic: it does not solve any goal. It is useful for defining other tacticals.

7.13.3 `Do` *num* *tactic*

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

7.13.4 $tactic_1$ `Orelse` $tactic_2$

The tactical $tactic_1$ `Orelse` $tactic_2$ tries to apply $tactic_1$ and, in case of a failure, applies $tactic_2$. It associates to the left.

7.13.5 `Repeat` *tactic*

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

7.13.6 $tactic_1$ `;` $tactic_2$

This tactic operator is a generalized composition for sequencing. The tactical $tactic_1$ `;` $tactic_2$ first applies $tactic_1$ and then applies $tactic_2$ to any subgoal generated by $tactic_1$. `;` associates to the left.

7.13.7 $tactic_0$`;` `[` $tactic_1$ `|` `...` `|` $tactic_n$ `]`

This tactic operator is a generalization of the precedent tactics operator. The tactical $tactic_0$ `;` `[` $tactic_1$ `|` `...` `|` $tactic_n$ `]` first applies $tactic_0$ and then applies $tactic_i$ to the i-th subgoal generated by $tactic_0$. It fails if n is not the exact number of remaining subgoals.

7.13.8 `Try` *tactic*

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

7.13.9 `First [` *tactic*$_0$ `|` ... `|` *tactic*$_n$ `]`

This tactic operator tries to apply the tactics *tactic*$_i$ with $i = 0 \dots n$, starting from $i = 0$, until one of them does not fail. It fails if all the tactics fail.

Error messages:

1. `No applicable tactic.`

7.13.10 `Solve [` *tactic*$_0$ `|` ... `|` *tactic*$_n$ `]`

This tactic operator tries to solve the current goal with the tactics *tactic*$_i$ with $i = 0 \dots n$, starting from $i = 0$, until one of them solves. It fails if no tactic can solve.

Error messages:

1. `Cannot solve the goal.`

7.13.11 `Info` *tactic*

This is not really a tactical. For elementary tactics, this is equivalent to *tactic*. For complex tactic like `Auto`, it displays the operations performed by the tactic.

7.13.12 `Abstract` *tactic*

From outside, typing `Abstract` *tactic* is the same that typing *tactic*. Internally it saves an auxiliary lemma called *ident*_`subproof`*n* where *ident* is the name of the current goal and *n* is chosen so that this is a fresh name.

This tactical is useful with tactics such `Omega` or `Discriminate` that generate big proof terms. With that tool the user can avoid the explosion at time of the `Save` command without having to cut "by hand" the proof in smaller lemmas.

Variants:

1. `Abstract` *tactic* `using` *ident*.
   Give explicitly the name of the auxiliary lemma.

7.14 Generation of induction principles with `Scheme`

The `Scheme` command is a high-level tool for generating automatically (possibly mutual) induction principles for given types and sorts. Its syntax follows the schema:

`Scheme` *ident*$_1$ `:= Induction for` *ident'*$_1$ `Sort` *sort*$_1$
`with`
 ...
`with` *ident*$_m$ `:= Induction for` *ident'*$_m$ `Sort` *sort*$_m$

ident'$_1$ ... *ident'*$_m$ are different inductive type identifiers belonging to the same package of mutual inductive definitions. This command generates *ident*$_1$... *ident*$_m$ to be mutually recursive definitions. Each term *ident*$_i$ proves a general principle of mutual induction for objects in type *term*$_i$.

Variants:

1. `Scheme` *ident*$_1$ `:= Minimality for` *ident'*$_1$ `Sort` *sort*$_1$
   `with`
   `...`
   `with` *ident*$_m$ `:= Minimality for` *ident'*$_m$ `Sort` *sort*$_m$
   Same as before but defines a non-dependent elimination principle more natural in case of inductively defined relations.

See also: 8.3

7.15 Simple tactic macros

A simple example has more value than a long explanation:

```
Coq < Tactic Definition Solve := Simpl; Intros; Auto.
Solve is defined
```

```
Coq < Tactic Definition ElimBoolRewrite b H1 H2 :=
Coq <   Elim b;
Coq <   [Intros; Rewrite H1; EAuto | Intros; Rewrite H2; EAuto ].
ElimBoolRewrite is defined
```

The tactics macros are synchronous with the Coq section mechanism: a `Tactic Definition` is deleted from the current environment when you close the section (see also 2.4) where it was defined. If you want that a tactic macro defined in a module is usable in the modules that require it, you should put it outside of any section.

The chapter 10 gives examples of more complex user-defined tactics.

Chapter 8

Detailed examples of tactics

This chapter presents detailed examples of certain tactics, to illustrate their behavior.

8.1 `Refine`

This tactic applies to any goal. It behaves like `Exact` with a big difference : the user can leave some holes (denoted by `?` or `(?::`*type*`)`) in the term. `Refine` will generate as many subgoals as they are holes in the term. The type of holes must be either synthesized by the system or declared by an explicit cast like `(?::nat->Prop)`. This low-level tactic can be useful to advanced users.

Example:

```
Coq < Inductive Option: Set := Fail : Option | Ok : bool->Option.

Coq < Definition get: (x:Option)~x=Fail->bool.
1 subgoal

  ============================
   (x:Option)~x=Fail->bool

Coq < Refine
Coq <   [x:Option]<[x:Option]~x=Fail->bool>Cases x of
Coq <          Fail   =>  ?
Coq <        | (Ok b) =>  [_:?]b end.
1 subgoal

  x : Option
  ============================
   ~Fail=Fail->bool

Coq < Intros;Absurd Fail=Fail;Trivial.
Subtree proved!

Coq < Defined.
```

8.2 `EApply`

Example: Assume we have a relation on `nat` which is transitive:

```
Coq < Variable R:nat->nat->Prop.

Coq < Hypothesis Rtrans : (x,y,z:nat)(R x y)->(R y z)->(R x z).

Coq < Variables n,m,p:nat.

Coq < Hypothesis Rnm:(R n m).

Coq < Hypothesis Rmp:(R m p).
```

Consider the goal `(R n p)` provable using the transitivity of `R`:

```
Coq < Goal (R n p).
```

The direct application of `Rtrans` with `Apply` fails because no value for `y` in `Rtrans` is found by `Apply`:

```
Coq < Apply Rtrans.
Error: generated subgoal (R n ?17) has metavariables in it
```

A solution is to rather apply `(Rtrans n m p)`.

```
Coq < Apply (Rtrans n m p).
2 subgoals

  ============================
   (R n m)
subgoal 2 is:
 (R m p)
```

More elegantly, `Apply Rtrans with y:=m` allows to only mention the unknown `m`:

```
Coq < Apply Rtrans with y:=m.
2 subgoals

  ============================
   (R n m)
subgoal 2 is:
 (R m p)
```

Another solution is to mention the proof of `(R x y)` in `Rtrans`...

```
Coq < Apply Rtrans with 1:=Rnm.
1 subgoal

  ============================
   (R m p)
```

... or the proof of `(R y z)`:

```
Coq < Apply Rtrans with 2:=Rmp.
1 subgoal

  ============================
   (R n m)
```

On the opposite, one can use `EApply` which postpone the problem of finding `m`. Then one can apply the hypotheses `Rnm` and `Rmp`. This instantiates the existential variable and completes the proof.

```
Coq < EApply Rtrans.
2 subgoals

  ============================
   (R n ?6)
subgoal 2 is:
 (R ?6 p)
```

```
Coq < Apply Rnm.
1 subgoal

  ============================
   (R m p)
```

```
Coq < Apply Rmp.
Subtree proved!
```

8.3 Scheme

Example 1: *Induction scheme for `tree` and `forest`*

The definition of principle of mutual induction for `tree` and `forest` over the sort `Set` is defined by the command:

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

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

```
Coq < Check tree_forest_rec.
tree_forest_rec
     : (P:(tree->Set); P0:(forest->Set))
        ((a:A; f:forest)(P0 f)->(P (node a f)))
        ->((b:B)(P0 (leaf b)))
        ->((t:tree)(P t)->(f:forest)(P0 f)->(P0 (cons t f)))
        ->(t:tree)(P t)
```

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

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

```
Coq < Check forest_tree_rec.
forest_tree_rec
     : (P:(tree->Set); P0:(forest->Set))
        ((a:A; f:forest)(P0 f)->(P (node a f)))
        ->((b:B)(P0 (leaf b)))
        ->((t:tree)(P t)->(f:forest)(P0 f)->(P0 (cons t f)))
        ->(f2:forest)(P0 f2)
```

Example 2: *Predicates* `odd` *and* `even` *on naturals*

Let `odd` and `even` be inductively defined as:

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

The following command generates a powerful elimination principle:

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

The type of `odd_even` for instance will be:

```
Coq < Check odd_even.
odd_even
     : (P,P0:(nat->Prop))
         ((n:nat)(even n)->(P0 n)->(P (S n)))
         ->(P0 O)
         ->((n:nat)(odd n)->(P n)->(P0 (S n)))
         ->(n:nat)(odd n)->(P n)
```

The type of `even_odd` shares the same premises but the conclusion is `(n:nat)(even n)->(Q n)`.

8.4 `Inversion`

Generalities about inversion

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

- we have an inconsistent instance of an inductive predicate in the local context of hypotheses. Thus, the current goal can be trivially proved by absurdity.
- we have a hypothesis that is an instance of an inductive predicate, and the instance has some variables whose constraints we would like to derive.

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

1. tactics for inverting an instance without stocking the inversion lemma in the context; this includes the tactics (`Dependent`) `Inversion` and (`Dependent`) `Inversion_clear`.
2. commands for generating and stocking in the context the inversion lemma corresponding to an instance; this includes `Derive` (`Dependent`) `Inversion` and `Derive` (`Dependent`) `Inversion_clear`.
3. tactics for inverting an instance using an already defined inversion lemma; this includes the tactic `Inversion ...using`.

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

Example 1: *Non-dependent inversion*
Let's consider the relation Le over natural numbers and the following variables:

```
Coq < Inductive Le : nat->nat->Set :=
Coq <   LeO : (n:nat)(Le O n)  |  LeS : (n,m:nat) (Le n m)-> (Le (S n) (S m)).

Coq < Variable P:nat->nat->Prop.

Coq < Variable Q:(n,m:nat)(Le n m)->Prop.
```

For example, consider the goal:

```
Coq < Show.
1 subgoal

  n : nat
  m : nat
  H : (Le (S n) m)
  ============================
   (P n m)
```

To prove the goal we may need to reason by cases on H and to derive that m is necessarily of the form $(S\ m_0)$ for certain m_0 and that $(Le\ n\ m_0)$. Deriving these conditions corresponds to prove that the only possible constructor of `(Le (S n) m)` is LeS and that we can invert the -> in the type of LeS. This inversion is possible because Le is the smallest set closed by the constructors LeO and LeS.

```
Coq < Inversion_clear  H.
1 subgoal

  n : nat
  m : nat
  m0 : nat
  H0 : (Le n m0)
  ============================
   (P n (S m0))
```

Note that m has been substituted in the goal for `(S m0)` and that the hypothesis `(Le n m0)` has been added to the context.

Sometimes it is interesting to have the equality `m=(S m0)` in the context to use it after. In that case we can use Inversion that does not clear the equalities:

```
Coq < Undo.

Coq < Inversion H.
1 subgoal

  n : nat
  m : nat
  H : (Le (S n) m)
  n0 : nat
```

```
  m0 : nat
  H0 : n0=n
  H2 : (S m0)=m
  H1 : (Le n m0)
  ============================
   (P n (S m0))
```

Example 2: *Dependent Inversion*
Let us consider the following goal:

```
Coq < Show.
1 subgoal

  n : nat
  m : nat
  H : (Le (S n) m)
  ============================
   (Q (S n) m H)
```

As `H` occurs in the goal, we may want to reason by cases on its structure and so, we would like inversion tactics to substitute `H` by the corresponding term in constructor form. Neither `Inversion` nor `Inversion_clear` make such a substitution. To have such a behavior we use the dependent inversion tactics:

```
Coq < Dependent Inversion_clear H.
1 subgoal

  n : nat
  m : nat
  m0 : nat
  l : (Le n m0)
  ============================
   (Q (S n) (S m0) (LeS n m0 l))
```

Note that `H` has been substituted by `(LeS n m0 l)` and `m` by `(S m0)`.

Example 3: *using already defined inversion lemmas*
For example, to generate the inversion lemma for the instance `(Le (S n) m)` and the sort `Prop` we do:

```
Coq < Derive Inversion_clear leminv with (n,m:nat)(Le (S n) m) Sort Prop.

Coq < Check leminv.
leminv
     : (n,m:nat; P:(nat->nat->Prop))
         ((m0:nat)(Le n m0)->(P n (S m0)))->(Le (S n) m)->(P n m)
```

Then we can use the proven inversion lemma:

```
Coq < Show.
1 subgoal

  n : nat
```

```
  m : nat
  H : (Le (S n) m)
  ============================
   (P n m)

Coq < Inversion H using leminv.
1 subgoal

  n : nat
  m : nat
  H : (Le (S n) m)
  ============================
   (m0:nat)(Le n m0)->(P n (S m0))
```

8.5 AutoRewrite

Here are two examples of `AutoRewrite` use. The first one (*Ackermann function*) shows actually a quite basic use where there is no conditional rewriting. The second one (*Mac Carthy function*) involves conditional rewritings and shows how to deal with them using the optional tactic of the `Hint Rewrite` command.

Example 1: *Ackermann function*

```
Coq < Require Arith.

Coq <
Coq < Variable Ack:nat->nat->nat.

Coq <
Coq < Axiom Ack0:(m:nat)(Ack (0) m)=(S m).

Coq < Axiom Ack1:(n:nat)(Ack (S n) (0))=(Ack n (1)).

Coq < Axiom Ack2:(n,m:nat)(Ack (S n) (S m))=(Ack n (Ack (S n) m)).

Coq < Hint Rewrite [ Ack0 Ack1 Ack2 ] in base0.

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

  ============================
   (Ack (3) (2))=(29)

Coq < AutoRewrite [ base0 ] using Try Reflexivity.
Subtree proved!
```

Example 2: *Mac Carthy function*

```
Coq < Require Omega.

Coq <
Coq < Variable g:nat->nat->nat.

Coq <
Coq < Axiom g0:(m:nat)(g (0) m)=m.
```

```
Coq < Axiom g1:
Coq <   (n,m:nat)(gt n (0))->(gt m (100))->(g n m)=(g (pred n) (minus m (10))).

Coq < Axiom g2:
Coq <   (n,m:nat)(gt n (0))->(le m (100))->(g n m)=(g (S n) (plus m (11))).

Coq < Hint Rewrite [ g0 g1 g2 ] in base1 using Omega.

Coq <
Coq < Lemma Resg0:(g (1) (110))=(100).
1 subgoal

  ============================
   (g (1) (110))=(100)

Coq < AutoRewrite [ base1 ] using Reflexivity Orelse Simpl.
Subtree proved!

Coq < Lemma Resg1:(g (1) (95))=(91).
1 subgoal

  ============================
   (g (1) (95))=(91)

Coq < AutoRewrite [ base1 ] using Reflexivity Orelse Simpl.
Subtree proved!
```

8.6 Quote

The tactic `Quote` allows to use Barendregt's so-called 2-level approach without writing any ML code. Suppose you have a language `L` of 'abstract terms' and a type `A` of 'concrete terms' and a function `f : L -> A`. If `L` is a simple inductive datatype and `f` a simple fixpoint, `Quote f` will replace the head of current goal by a convertible term of the form `(f t)`. `L` must have a constructor of type: `A -> L`.

Here is an example:

```
Coq < Require Quote.

Coq < Parameters A,B,C:Prop.
A is assumed
B is assumed
C is assumed

Coq < Inductive Type formula :=
Coq < | f_and : formula -> formula -> formula (* binary constructor *)
Coq < | f_or : formula -> formula -> formula
Coq < | f_not : formula -> formula           (* unary constructor *)
Coq < | f_true : formula                     (* 0-ary constructor *)
Coq < | f_const : Prop -> formula            (* contructor for constants *).
formula is defined
formula_rect is defined
formula_ind is defined
formula_rec is defined

Coq <
```

```
Coq < Fixpoint interp_f [f:formula] : Prop :=
Coq <   Cases f of
Coq <    | (f_and f1 f2) => (interp_f f1)/\(interp_f f2)
Coq <    | (f_or f1 f2) => (interp_f f1)\/(interp_f f2)
Coq <    | (f_not f1) => ~(interp_f f1)
Coq <    | f_true => True
Coq <    | (f_const c) => c
Coq <   end.
interp_f is recursively defined
```

```
Coq < Goal A/\(A\/True)/\~B/\(A <-> A).
1 subgoal

  ============================
   A/\(A\/True)/\~B/\(A<->A)
```

```
Coq < Quote interp_f.
1 subgoal

  ============================
   (interp_f
     (f_and (f_const A)
       (f_and (f_or (f_const A) f_true)
         (f_and (f_not (f_const B)) (f_const A<->A)))))
```

The algorithm to perform this inversion is: try to match the term with right-hand sides expression of `f`. If there is a match, apply the corresponding left-hand side and call yourself recursively on sub-terms. If there is no match, we are at a leaf: return the corresponding constructor (here `f_const`) applied to the term.

Error messages:

1. `Quote: not a simple fixpoint`
   Happens when `Quote` is not able to perform inversion properly.

8.6.1 Introducing variables map

The normal use of `Quote` is to make proofs by reflection: one defines a function `simplify : formula -> formula` and proves a theorem `simplify_ok: (f:formula)(interp_f (simplify f)) -> (interp_f f)`. Then, one can simplify formulas by doing:

```
    Quote interp_f.
Apply simplify_ok.
Compute.
```

But there is a problem with leafs: in the example above one cannot write a function that implements, for example, the logical simplifications $A \wedge A \to A$ or $A \wedge \neg A \to$ `False`. This is because the Prop is impredicative.

It is better to use that type of formulas:

```
Coq < Inductive Set formula :=
Coq < | f_and : formula -> formula -> formula
Coq < | f_or : formula -> formula -> formula
```

```
Coq < | f_not : formula -> formula
Coq < | f_true : formula
Coq < | f_atom : index -> formula.
formula is defined
formula_rect is defined
formula_ind is defined
formula_rec is defined
```

index is defined in module Quote. Equality on that type is decidable so we are able to simplify $A \wedge A$ into A at the abstract level.

When there are variables, there are bindings, and Quote provides also a type (varmap A) of bindings from index to any set A, and a function varmap_find to search in such maps. The interpretation function has now another argument, a variables map:

```
Coq <  Fixpoint interp_f [vm:(varmap Prop); f:formula] : Prop :=
Coq <   Cases f of
Coq <    | (f_and f1 f2) => (interp_f vm f1)/\(interp_f vm f2)
Coq <    | (f_or f1 f2) => (interp_f vm f1)\/(interp_f vm f2)
Coq <    | (f_not f1) => ~(interp_f vm f1)
Coq <    | f_true => True
Coq <    | (f_atom i) => (varmap_find True i vm)
Coq <   end.
interp_f is recursively defined
```

Quote handles this second case properly:

```
Coq < Goal A/\(B\/A)/\(A\/~B).
1 subgoal

  ============================
   A/\(B\/A)/\(A\/~B)

Coq < Quote interp_f.
1 subgoal

  ============================
   (interp_f
     (Node_vm B (Node_vm A (Empty_vm Prop) (Empty_vm Prop))
       (Empty_vm Prop))
     (f_and (f_atom (Left_idx End_idx))
       (f_and (f_or (f_atom End_idx) (f_atom (Left_idx End_idx)))
         (f_or (f_atom (Left_idx End_idx)) (f_not (f_atom End_idx))))))
```

It builds vm and t such that (f vm t) is convertible with the conclusion of current goal.

8.6.2 Combining variables and constants

One can have both variables and constants in abstracts terms; that is the case, for example, for the Ring tactic (chapter 18). Then one must provide to Quote a list of *constructors of constants*. For example, if the list is [O S] then closed natural numbers will be considered as constants and other terms as variables.

Example:

```
Coq < Inductive Type formula :=
Coq < | f_and : formula -> formula -> formula
Coq < | f_or : formula -> formula -> formula
Coq < | f_not : formula -> formula
Coq < | f_true : formula
Coq < | f_const : Prop -> formula            (* constructor for constants *)
Coq < | f_atom : index -> formula.           (* constructor for variables *)

Coq <
Coq < Fixpoint interp_f [vm:(varmap Prop); f:formula] : Prop :=
Coq <   Cases f of
Coq <    | (f_and f1 f2) => (interp_f vm f1)/\(interp_f vm f2)
Coq <    | (f_or f1 f2) => (interp_f vm f1)\/(interp_f vm f2)
Coq <    | (f_not f1) => ~(interp_f vm f1)
Coq <    | f_true => True
Coq <    | (f_const c) => c
Coq <    | (f_atom i) => (varmap_find True i vm)
Coq <   end.

Coq <
Coq < Goal A/\(A\/True)/\~B/\(C<->C).

Coq < Quote interp_f [A B].
1 subgoal

  ============================
   (interp_f (Node_vm C<->C (Empty_vm Prop) (Empty_vm Prop))
     (f_and (f_const A)
       (f_and (f_or (f_const A) f_true)
         (f_and (f_not (f_const B)) (f_atom End_idx)))))

Coq < Undo. Quote interp_f [B C iff].
1 subgoal

  ============================
   A/\(A\/True)/\~B/\(C<->C)
1 subgoal

  ============================
   (interp_f (Node_vm A (Empty_vm Prop) (Empty_vm Prop))
     (f_and (f_atom End_idx)
       (f_and (f_or (f_atom End_idx) f_true)
         (f_and (f_not (f_const B)) (f_const C<->C)))))
```

Warning: Since function inversion is undecidable in general case, don't expect miracles from it!

See also: comments of source file `tactics/contrib/polynom/quote.ml`

See also: the tactic `Ring` (chapter 18)

Part III

User extensions

Chapter 9

Syntax extensions

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. As in most compilers, there is an intermediate structure called *Abstract Syntax Tree* (AST). Parsing a term is done in two steps[1]:

1. An AST is build from the input (a stream of tokens), following grammar rules. This step consists in deciding whether the input belongs to the language or not. If it is, the parser transforms the expression into an AST. If not, this is a syntax error. An expression belongs to the language if there exists a sequence of grammar rules that recognizes it. This task is delegated to Camlp4. See the Reference Manual [29] for details on the parsing technology. The transformation to AST is performed by executing successively the *actions* bound to these rules.

2. The AST is translated into the internal representation of commands and terms. At this point, we detect unbound variables and determine the exact section-path of every global value. Then, the term may be typed, computed, ...

The printing process is the reverse: commands or terms are first translated into AST's, and then the pretty-printer translates this AST into a printing orders stream, according to printing rules.

In Coq, only the translations between AST's and the concrete representation are extendable. One can modify the set of grammar and printing rules, but one cannot change the way AST's are interpreted in the internal level.

In the following section, we describe the syntax of AST expressions, involved in both parsing and printing. The next two sections deal with extendable grammars and pretty-printers.

9.1 Abstract syntax trees (AST)

The AST expressions are conceptually divided into two classes: *constructive expressions* (those that can be used in parsing rules) and *destructive expressions* (those that can be used in pretty printing rules). In the following we give the concrete syntax of expressions and some examples of their usage.

[1]We omit the lexing step, which simply translates a character stream into a token stream. If this translation fails, this is a *Lexical error*.

ast	::=	*meta*	(metavariable)
	|	*ident*	(variable)
	|	*integer*	(positive integer)
	|	*string*	(quoted string)
	|	*path*	(section-path)
	|	`{` *ident* `}`	(identifier)
	|	`[` *name* `]` *ast*	(abstraction)
	|	`(` *ident* *[ast ... ast]* `)`	(application node)
	|	`(` *special-tok* *meta* `)`	(special-operator)
	|	`'` *ast*	(quoted ast)
	|	*quotation*	
meta	::=	`$`*ident*	
path	::=	`#`*ident* ... `#`*ident* `.` *kind*	
kind	::=	`cci` | `fw` | `obj`	
name	::=	`<>` | *ident* | *meta*	
special-tok	::=	`$LIST` \| `$VAR` | `$NUM` \| `$STR` | `$PATH` \| `$ID`	
quotation	::=	`<<` *meta-constr* `>>`	
	|	`<:constr:<` *meta-constr* `>>`	
	|	`<:vernac:<` *meta-vernac* `>>`	
	|	`<:tactic:<` *meta-tactic* `>>`	

Figure 9.1: Syntax of AST expressions

The BNF grammar *ast* in Fig. 9.1 defines the syntax of both constructive and destructive expressions. The lexical conventions are the same as in section 1.1. Let us first describe the features common to constructive and destructive expressions.

Atomic AST

An atomic AST can be either a variable, a natural number, a quoted string, a section path or an identifier. They are the basic components of an AST.

Metavariable

Metavariables are used to perform substitutions in constructive expressions: they are replaced by their value in a given environment. They are also involved in the pattern matching operation: metavariables in destructive patterns create new bindings in the environment.

As we will see later, metavariables may denote an AST or an AST list (when used with the `$LIST` special token). So, we introduce two types of variables: `ast` and `ast list`. The type of variables is checked statically: an expression referring to undefined metavariables, or using metavariables with an inappropriate type, will be rejected.

Application node

Note that the AST syntax is rather general, since application nodes may be labelled by an arbitrary identifier (but not a metavariable), and operators have no fixed arity. This enables the extensibility of the system.

Nevertheless there are some application nodes that have some special meaning for the system. They are build on `(`*special-tok meta*`)`, and cannot be confused with regular nodes since *special-tok* begins with a `$`. There is a description of these nodes below.

Abstraction

The equality on AST's is the α-conversion, i.e. two AST's are equal if only they are the same up to renaming of bound variables (thus, `[x]x` is equal to `[y]y`). This makes the difference between variables and identifiers clear: the former may be bound by abstractions, whereas identifiers cannot be bound. To illustrate this, `[x]x` and `[y]y` are equal and `[x]{x}` is equal to `[y]{x}`, but not to `[y]{y}`.

The binding structure of AST is used to represent the binders in the terms of Coq: the product `(x:$A)$B` is mapped to the AST `(PROD $A [x]$B)`, whereas the non dependent product `$A->$B` is mapped to `(PROD $A [<>]$B)` (`[<>]t` is an anonymous abstraction).

Metavariables can appear in abstractions. In that case, the value of the metavariable must be a variable (or a list of variables). If not, a run-time error is raised.

Quoted AST

The `'`t construction means that the AST t should not be interpreted at all. The main effect is that metavariables occurring in it cannot be substituted or considered as binding in patterns.

Quotations

The non terminal symbols *meta-constr*, *meta-vernac* and *meta-tactic* stand, respectively, for the syntax of CIC terms, vernacular phrases and tactics. The prefix *meta-* is just to emphasize that the expression may refer to metavariables.

Indeed, if the AST to generate corresponds to a term that already has a syntax, one can call a grammar to parse it and to return the AST result. For instance, `<<(eq ? $f $g)>>` denotes the AST which is the application (in the sense of CIC) of the constant `eq` to three arguments. It is coded as an AST node labelled `APPLIST` with four arguments.

This term is parsable by `constr:constr` grammar. This grammar is invoked on this term to generate an AST by putting the term between "<<" and ">>".

We can also invoke the initial grammars of several other predefined entries (see section 9.2.1 for a description of these grammars).

- `<:constr:< t >>` parses `t` with `constr:constr` grammar(terms of CIC).
- `<:vernac:< t >>` parses `t` with `vernac:vernac` grammar (vernacular commands).
- `<:tactic:< t >>` parses `t` with `tactic:tactic` grammar (tactic expressions).
- `<< t >>` parses `t` with the default quotation (that is, `constr:constr`). It is the same as `<:constr:< t >>`.

Warning: One cannot invoke other grammars than those described.

Special operators in constructive expressions

The expressions `($LIST $x)` injects the AST list variable `$x` in an AST position. For example, an application node is composed of an identifier followed by a list of AST's that are glued together. Each of these expressions must denote an AST. If we want to insert an AST list, one has to use the `$LIST` operator. Assume the variable `$idl` is bound to the list `[x y z]`, the expression `(Intros ($LIST $idl) a b c)` will build the AST `(Intros x y z a b c)`. Note that `$LIST` does not occur in the result.

Since we know the type of variables, the `$LIST` is not really necessary. We enforce this annotation to stress on the fact that the variable will be substituted by an arbitrary number of AST's.

The other special operators (`$VAR`, `$NUM`, `$STR`, `$PATH` and `$ID`) are forbidden.

Special operators in destructive expressions (AST patterns)

A pattern is an AST expression, in which some metavariables can appear. In a given environment a pattern matches any AST which is equal (w.r.t α-conversion) to the value of the pattern in an extension of the current environment. The result of the matching is precisely this extended environment. This definition allows non-linear patterns (i.e. patterns in which a variable occurs several times).

For instance, the pattern `(PAIR $x $x)` matches any AST which is a node labelled `PAIR` applied to two identical arguments, and binds this argument to `$x`. If `$x` was already bound, the arguments must also be equal to the current value of `$x`.

The "wildcard pattern" `$_` is not a regular metavariable: it matches any term, but does not bind any variable. The pattern `(PAIR $_ $_)` matches any `PAIR` node applied to two arguments.

The `$LIST` operator still introduces list variables. Typically, when a metavariable appears as argument of an application, one has to say if it must match *one* argument (binding an AST variable), or *all* the arguments (binding a list variable). Let us consider the patterns `(Intros $id)` and `(Intros ($LIST $idl))`. The former matches nodes with *exactly* one argument, which is bound in the AST variable `$id`. On the other hand, the latter pattern matches any AST node labelled `Intros`, and it binds the *list* of its arguments to the list variable `$idl`. The `$LIST` pattern must be the last item of a list pattern, because it would make the pattern matching operation more complicated and less efficient. The pattern `(Intros ($LIST $idl) $lastid)` is not accepted.

The other special operators allows checking what kind of leaf we are destructing:

- `$VAR` matches only variables
- `$NUM` matches natural numbers
- `$STR` matches quoted strings
- `$PATH` matches section-paths
- `$ID` matches identifiers

For instance, the pattern `(DO ($NUM $n) $tc)` matches `(DO 5 (Intro))`, and creates the bindings (`$n`,5) and (`$tc`,`(Intro)`). The pattern matching would fail on `(DO "5" (Intro))`.

grammar	::=	`Grammar` *entry gram-entry* `with` ... `with` *gram-entry*
entry	::=	*ident*
gram-entry	::=	*rule-name* [`:` *entry-type*] `:=` [*production* | ... | *production*]
entry-type	::=	`ast` | `ast list` | `constr` | `tactic` | `vernac`
production	::=	*rule-name* `[` [*prod-item* ... *prod-item*] `]` `->` *action*
rule-name	::=	*ident*
prod-item	::=	*string*
	|	[*entry* `:`] *entry-name* [`(` *meta* `)`]
action	::=	`[` [*ast-quote* ... *ast-quote*] `]`
	|	`let` *pattern* `=` *action* `in` *action*
	|	`case` *action* [`:` *entry-type*] `of` [*case* | ... | *case*] `esac`
case	::=	[*pattern* ... *pattern*] `->` *action*
pattern	::=	*ast*

Figure 9.2: Syntax of the grammar extension command

9.2 Extendable grammars

Grammar rules can be added with the `Grammar` command. This command is just an interface towards Camlp4, providing the semantic actions so that they build the expected AST. A simple grammar command has the following syntax:

`Grammar` *entry nonterminal* `:=` *rule-name LMP* `->` *action* `.`

The components have the following meaning:

- a grammar name: defined by a parser entry and a non-terminal. Non-terminals are packed in an *entry* (also called universe). One can have two non-terminals of the same name if they are in different entries. A non-terminal can have the same name as its entry.

- a rule (sometimes called production), formed by a name, a left member of production and an action, which generalizes constructive expressions.

The exact syntax of the `Grammar` command is defined in Fig. 9.2 where non terminal *ast-quote* is one of `ast`, `constr`, `tactic` or `vernac`, depending on the entry type.

It is possible to extend a grammar with several rules at once.

$$\begin{array}{lcl}\texttt{Grammar}\ entry\ nonterminal & \texttt{:=} & production_1 \\ & | & \vdots \\ & | & production_n\ \texttt{.}\end{array}$$

Productions are entered in reverse order (i.e. $production_n$ before $production_1$), so that the first rules have priority over the last ones. The set of rules can be read as an usual pattern matching.

Also, we can extend several grammars of a given universe at the same time. The order of non-terminals does not matter since they extend different grammars.

$$
\begin{array}{llllll}
\texttt{Grammar} & entry & nonterminal_1 & \texttt{:=} & production^1_1 \\
 & & & | & \vdots \\
 & & & | & production^1_{n_1} \\
 & \texttt{with} & \vdots & & \\
 & \texttt{with} & nonterminal_p & \texttt{:=} & production^p_1 \\
 & & & | & \vdots \\
 & & & | & production^p_{n_p}\ \texttt{.}
\end{array}
$$

9.2.1 Grammar entries

Let us describe the four predefined entries. Each of them (except `prim`) possesses an initial grammar for starting the parsing process.

- `prim` : it is the entry of the primitive grammars. Most of them cannot be defined by the extendable grammar mechanism. They are encoded inside the system. The entry contains the following non-terminals:
  - `var` : variable grammar. Parses an identifier and builds an AST which is a variable.
  - `ident` : identifier grammar. Parses an identifier and builds an AST which is an identifier such as `{x}`.
  - `number` : number grammar. Parses a positive integer.
  - `string` : string grammar. Parses a quoted string.
  - `path` : section path grammar.
  - `ast` : AST grammar.
  - `astpat` : AST pattern grammar.
  - `astact` : action grammar.

  The primitive grammars are used as the other grammars; for instance the variables of terms are parsed by `prim:var($id)`.

- `constr` : it is the term entry. It allows to have a pretty syntax for terms. Its initial grammar is `constr:constr`. This entry contains several non-terminals, among them `constr0` to `constr10` which stratify the terms according to priority levels (`0` to `10`). These priority levels allow us also to specify the order of associativity of operators.

- `vernac` : it is the vernacular command entry, with `vernac vernac` as initial grammar. Thanks to it, the developers can define the syntax of new commands they add to the system. As to users, they can change the syntax of the predefined vernacular commands.

- `tactic` : it is the tactic entry with `tactics:tactic` as initial grammar. This entry allows to define the syntax of new tactics or redefine the syntax of existing tactics.

The user can define new entries and new non-terminals, using the grammar extension command. A grammar does not have to be explicitly defined. But the grammars in the left member of rules must all be defined, possibly by the current grammar command. It may be convenient to define an empty grammar, just so that it may be called by other grammars, and extend this empty grammar later. Assume that the `constr:constr13` does not exist. The next command defines it with zero productions.

```
Coq < Grammar constr constr13 := .
```

The grammars of new entries do not have an initial grammar. To use them, they must be called (directly or indirectly) by grammars of predefined entries. We give an example of a (direct) call of the grammar `newentry:nonterm` by `constr:constr`. This following rule allows to use the syntax `a&b` for the conjunction `a/\b`. Note that since we extend a rule of universe `constr`, the command quotation is used on the right-hand side of the second rule.

```
Coq < Grammar newentry nonterm :=
Coq <   ampersand [ "&" constr:constr($c) ] -> [$c].
```

```
Coq < Grammar constr constr :=
Coq <   new_and [ constr8($a) newentry:nonterm($b) ] -> [$a/\$b].
```

9.2.2 Left member of productions (LMP)

A LMP is composed of a combination of terminals (enclosed between double quotes) and grammar calls specifying the entry. It is enclosed between "`[`" and "`]`". The empty LMP, represented by `[ ]`, corresponds to ϵ in formal language theory.

A grammar call is done by `entry:nonterminal($id)` where:

- `entry` and `nonterminal` specifies the entry of the grammar, and the non-terminal.
- `$id` is a metavariable that will receive the AST or AST list resulting from the call to the grammar.

The elements `entry` and `$id` are optional. The grammar entry can be omitted if it is the same as the entry of the non-terminal we are extending. Also, `$id` is omitted if we do not want to get back the AST result. Thus a grammar call can be reduced to a non-terminal.

Each terminal must contain exactly one token. This token does not need to be already defined. If not, it will be automatically added. Nevertheless, any string cannot be a token (e.g. blanks should not appear in tokens since parsing would then depend on indentation). We introduce the notion of *valid token*, as a sequence, without blanks, of characters taken from the following list:

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

that do not start with a character from

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

When an LMP is used in the parsing process of an expression, it is analyzed from left to right. Every token met in the LMP should correspond to the current token of the expression. As for the grammars calls, they are performed in order to recognize parts of the initial expression.

Warning: Unlike destructive expressions, if a variable appears several times in the LMP, the last binding hides the previous ones. Comparison can be performed only in the actions.

Example 1: *Defining a syntax for inequality*

The rule below allows us to use the syntax `t1#t2` for the term `~t1=t2`.

```
Coq < Grammar constr constr1 :=
Coq <   not_eq [ constr0($a) "#" constr0($b) ] -> [ ~$a=$b ].
```

The level 1 of the grammar of terms is extended with one rule named `not_eq`. When this rule is selected, its LMP calls the grammar `constr:constr0`. This grammar recognizes a term that it binds to the metavariable `$a`. Then it meets the token "`#`" and finally it calls the grammar `constr:constr0`. This grammar returns the recognized term in `$b`. The action constructs the term `~$a=$b`.

For instance, let us give the statement of the symmetry of `#`:

```
Coq < Goal (A:Set)(a,b:A) a#b -> b#a.
1 subgoal

  ============================
   (A:Set; a,b:A)~a=b->~b=a
```

This shows that the system understood the grammar extension. Nonetheless, since no special printing command was given, the goal is displayed using the usual syntax for negation and equality. One can force `~a=b` to be printed `a#b` by giving pretty-printing rules. This is explained in section 9.3.

Warning: Metavariables are identifiers preceded by the "`$`" symbol. They cannot be replaced by identifiers. For instance, if we enter a rule with identifiers and not metavariables, the identifiers are assumed to be global names (what raises a warning if no global name is denoted by these identifiers).

```
Coq < Grammar constr constr1 :=
Coq <   not_eq [ constr0($a) "#" constr0($b) ] -> [~(a=b)].
<W> Grammar extension: some rule has been masked
Warning: Could not globalize a
Warning: Could not globalize b
```

Example 2: *Redefining vernac commands*

Thanks to the following rule, "`|- term.`" will have the same effect as "`Goal term.`".

```
Coq < Grammar vernac vernac :=
Coq <   thesis [ "|" "-" constr:constr($term) "." ]
Coq <            -> [Goal $term.].
```

This rule allows putting blanks between the bar and the dash, as in

```
Coq < | - (A:Prop)A->A.
1 subgoal

  ============================
   (A:Prop)A->A
```

Assuming the previous rule has not been entered, we can forbid blanks with a rule that declares "`|-`" as a single token:

```
Coq < Grammar vernac vernac :=
Coq <   thesis [ "|-" constr:constr($term) "." ]
Coq <           -> [Goal $term.].

Coq < | - (A:Prop)A->A.
Toplevel input, characters 0-1
> | - (A:Prop)A->A.
> ^
Syntax error: illegal begin of vernac
```

If both rules were entered, we would have three tokens |, - and |-. The lexical ambiguity on the string |- is solved according to the longest match rule (see lexical conventions page 25), i.e. |- would be one single token. To enforce the use of the first rule, a blank must be inserted between the bar and the dash[2].

Remark: The vernac commands should always be terminated by a period. When a syntax error is detected, the top-level discards its input until it reaches a period token, and then resumes parsing.

Example 3: *Redefining tactics*

We can give names to repetitive tactic sequences. Thus in this example "`IntSp`" will correspond to the tactic `Intros` followed by `Split`.

```
Coq < Grammar tactic simple_tactic :=
Coq <   intros_split [ "IntSp" ] -> [Intros; Split].
```

Let us check that this works.

```
Coq < Goal (A,B:Prop)A/\B -> B/\A.
1 subgoal

  ============================
   (A,B:Prop)A/\B->B/\A

Coq < IntSp.
2 subgoals

  A : Prop
  B : Prop
  H : A/\B
  ============================
   B
subgoal 2 is:
 A
```

Note that the same result can be obtained in a simpler way with `Tactic Definition` (see chapter 10).

Example 4: *Priority, left and right associativity of operators*

The disjunction has a higher priority than conjunction. Thus `A/\B\/C` will be parsed as `(A/\B)\/C` and not as `A/\(B\/C)`. The priority is done by putting the rule for the disjunction in a higher level than that of conjunction: conjunction is defined in the non-terminal `constr6` and disjunction in `constr7` (see file `Logic.v` in the library). Notice that the character "\" must be doubled (see lexical conventions for quoted strings on page 25).

[2]It turns out that "|-" is already a token defined for other purposes, then the first rule cannot parse "|- (A:Prop)A->A" and indeed requires the insertion of a blank

```
Coq < Grammar constr constr6 :=
Coq <   and [ constr5($c1) "/\\" constr6($c2) ] -> [(and $c1 $c2)].

Coq < Grammar constr constr7 :=
Coq <   or  [ constr6($c1) "\\/" constr7($c2) ] -> [(or $c1 $c2)].
```

Thus conjunction and disjunction associate to the right since in both cases the priority of the right term (resp. `constr6` and `constr7`) is higher than the priority of the left term (resp. `constr5` and `constr6`). The left member of a conjunction cannot be itself a conjunction, unless you enclose it inside parenthesis.

The left associativity is done by calling recursively the non-terminal. Camlp4 deals with this recursion by first trying the non-left-recursive rules. Here is an example taken from the standard library, defining a syntax for the addition on integers:

```
Coq < Grammar znatural expr :=
Coq <   expr_plus [ expr($p) "+" expr($c) ] -> [(Zplus $p $c)].
```

9.2.3 Actions

Every rule should generate an AST corresponding to the syntactic construction that it recognizes. This generation is done by an action. Thus every rule is associated to an action. The syntax has been defined in Fig. 9.2. We give some examples.

Simple actions

A simple action is an AST enclosed between "`[`" and "`]`". It simply builds the AST by interpreting it as a constructive expression in the environment defined by the LMP. This case has been illustrated in all the previous examples. We will later see that grammars can also return AST lists.

Local definitions

When an action should generate a big term, we can use `let` *pattern* = $action_1$ `in` $action_2$ expressions to construct it progressively. The action $action_1$ is first computed, then it is matched against *pattern* which may bind metavariables, and the result is the evaluation of $action_2$ in this new context.

Example 5:
From the syntax `t1*+t2`, we generate the term `(plus (plus t1 t2) (mult t1 t2))`.

```
Coq < Grammar constr constr1 :=
Coq <   mult_plus [ constr0($a) "*" "+" constr0($b) ]
Coq <       -> let $p1=[(plus $a $b)] in
Coq <          let $p2=[(mult $a $b)] in
Coq <            [(plus $p1 $p2)].
```

Let us give an example with this syntax:

```
Coq < Goal (O*+O)=O.
1 subgoal

  ============================
   (plus (plus O O) (mult O O))=O
```

Conditional actions

We recall the syntax of conditional actions:

$$\texttt{case}\ \mathit{action}\ \texttt{of}\ \mathit{pattern}_1\ \texttt{->}\ \mathit{action}_1\ \mid \cdots \mid\ \mathit{pattern}_n\ \texttt{->}\ \mathit{action}_n\ \texttt{esac}$$

The action to execute is chosen according to the value of *action*. The matching is performed from left to right. The selected action is the one associated to the first pattern that matches the value of *action*. This matching operation will bind the metavariables appearing in the selected pattern. The pattern matching does need being exhaustive, and no warning is emitted. When the pattern matching fails a message reports in which grammar rule the failure happened.

Example 6: *Overloading the "+" operator*
The internal representation of an expression such as `A+B` depends on the shape of `A` and `B`:

- `{P}+{Q}` uses `sumbool`
- otherwise, `A+{Q}` uses `sumor`
- otherwise, `A+B` uses `sum`.

The trick is to build a temporary AST: `{A}` generates the node `(SQUASH A)`. When we parse `A+B`, we remove the `SQUASH` in `A` and `B`:

```
Coq < Grammar constr constr1: ast :=
Coq <   squash [ "{" lconstr($lc) "}" ] -> [(SQUASH $lc)].
```

```
Coq < Grammar constr lassoc_constr4 :=
Coq <   squash_sum
Coq <   [ lassoc_constr4($c1) "+" lassoc_constr4($c2) ] ->
Coq <       case [$c2] of
Coq <         (SQUASH $T2) ->
Coq <             case [$c1] of
Coq <               (SQUASH $T1) -> [(sumbool $T1 $T2)]
Coq <             | $_           -> [(sumor $c1 $T2)]
Coq <             esac
Coq <       | $_           -> [(sum $c1 $c2)]
Coq <       esac.
```

The first rule is casted with type ast, because the produced term cannot be reached by the input syntax. On the other hand, the second command has (implicit) type `constr`, so the right hand side is parsed with the term parser.
The problem is that sometimes, the intermediate `SQUASH` node cannot re-shaped, then we have a very specific error:

```
Coq < Check {True}.
Toplevel input, characters 6-12
> Check {True}.
>       ^^^^^^
Error: Ill-formed specification
```

Example 7: *Comparisons and non-linear patterns*
The patterns may be non-linear: when an already bound metavariable appears in a pattern, the value yielded by the pattern matching must be equal, up to renaming of bound variables, to the current value. Note that this does not apply to the wildcard `$_`. For example, we can compare two arguments:

```
Coq < Grammar constr constr10 :=
Coq <   refl_equals [ constr9($c1) "||" constr9($c2) ] ->
Coq <               case [$c1] of $c2 -> [(refl_equal ? $c2)] esac.
```

```
Coq < Check ([x:nat]x || [y:nat]y).
(refl_equal nat->nat [y:nat]y)
     : ([y:nat]y)=([y:nat]y)
```

The metavariable `$c1` is bound to `[x:nat]x` and `$c2` to `[y:nat]y`. Since these two values are equal, the pattern matching succeeds. It fails when the two terms are not equal:

```
Coq < Check ([x:nat]x || [z:bool]z).
Toplevel input, characters 7-28
> Check ([x:nat]x || [z:bool]z).
>        ^^^^^^^^^^^^^^^^^^^^^
Error: during interpretation of grammar rule refl_equals,
 Grammar case failure. The ast (LAMBDALIST (QUALID nat) [x](QUALID x))
 does not match any of the patterns : $c2
 with constraints :
  $c1 = (LAMBDALIST (QUALID nat) [x](QUALID x))
  $c2 = (LAMBDALIST (QUALID bool) [z](QUALID z))
```

9.2.4 Grammars of type `ast list`

Assume we want to define an non-terminal `ne_identarg_list` that parses an non-empty list of identifiers. If the grammars could only return AST's, we would have to define it this way:

```
Coq < Grammar tactic my_ne_ident_list : ast :=
Coq <   ident_list_cons [ identarg($id) my_ne_ident_list($l) ] ->
Coq <     case [$l] of
Coq <       (IDENTS ($LIST $idl)) -> [(IDENTS $id ($LIST $idl))]
Coq <     esac
Coq < | ident_list_single [ identarg($id) ] -> [(IDENTS $id)].
```

But it would be inefficient: every time an identifier is read, we remove the "boxing" operator `IDENTS`, and put it back once the identifier is inserted in the list.

To avoid these awkward trick, we allow grammars to return AST lists. Hence grammars have a type (`ast` or `ast list`), just like AST's do. Type-checking can be done statically.

The simple actions can produce lists by putting a list of constructive expressions one beside the other. As usual, the `$LIST` operator allows to inject AST list variables.

```
Coq < Grammar tactic ne_identarg_list : ast list :=
Coq <   ne_idl_cons [ identarg($id) ne_identarg_list($idl) ]
Coq <                   -> [ $id ($LIST $idl) ]
Coq < | ne_idl_single [ identarg($id) ] -> [ $id ].
```

Note that the grammar type must be recalled in every extension command, or else the system could not discriminate between a single AST and an AST list with only one item. If omitted, the default type depends on the universe name. The following command fails because the non-terminal `ne_identarg_list` is already defined with type `ast list` but the `Grammar` command header assumes its type is `ast`.

```
Coq < Grammar tactic ne_identarg_list :=
Coq <   list_two [ identarg($id1) identarg($id2) ] -> [ $id1 ; $id2 ].
Toplevel input, characters 15-31
> Grammar tactic ne_identarg_list :=
>                ^^^^^^^^^^^^^^^^
Error: Entry tactic:ne_identarg_list already exists with another type
```

All rules of a same grammar must have the same type. For instance, the following rule is refused because the `constr:constr1` grammar has been already defined with type `Ast`, and cannot be extended with a rule returning AST lists.

```
Coq < Grammar constr constr1 :=
Coq <   carret_list [ constr0($c1) "^" constr0($c2)] -> [ $c1 $c2 ].
Toplevel input, characters 82-85
>   carret_list [ constr0($c1) "^" constr0($c2)] -> [ $c1 $c2 ].
>                                                        ^^^
Syntax error: ']' expected after [default_action_parser] (in [action])
```

9.2.5 Limitations

The extendable grammar mechanism have four serious limitations. The first two are inherited from Camlp4.

- Grammar rules are factorized syntactically: Camlp4 does not try to expand non-terminals to detect further factorizations. The user must perform the factorization himself.
- The grammar is not checked to be *LL(1)* when adding a rule. If it is not LL(1), the parsing may fail on an input recognized by the grammar, because selecting the appropriate rule may require looking several tokens ahead. Camlp4 always selects the most recent rule (and all those that factorize with it) accepting the current token.
- There is no command to remove a grammar rule. However there is a trick to do it. It is sufficient to execute the "`Reset`" command on a constant defined before the rule we want to remove. Thus we retrieve the state before the definition of the constant, then without the grammar rule. This trick does not apply to grammar extensions done in Objective Caml.
- Grammar rules defined inside a section are automatically removed after the end of this section: they are available only inside it.

The command `Print Grammar` prints the rules of a grammar. It is displayed by Camlp4. So, the actions are not printed, and the recursive calls are printed `SELF`. It is sometimes useful if the user wants to understand why parsing fails, or why a factorization was not done as expected.

```
Coq < Print Grammar constr constr8.
[ LEFTA
  [ Constr.constr7; "<->"; SELF
  | Constr.constr7; "->"; SELF
  | Constr.constr7 ] ]
```

Getting round the lack of factorization

The first limitation may require a non-trivial work, and may lead to ugly grammars, hardly extendable. Sometimes, we can use a trick to avoid these troubles. The problem arises in the Gallina syntax, to make Camlp4 factorize the rules for application and product. The natural grammar would be:

```
Coq < Grammar constr constr0 : ast :=
Coq <   parenthesis [ "(" constr10($c) ")" ] -> [$c]
Coq < | product [ "(" prim:var($id) ":" constr($c1) ")" constr0($c2) ] ->
Coq <             [(PROD $c1 [$id]$c2)]
Coq < with constr10 : ast :=
Coq <   application [ constr9($c1) ne_constr_list($lc) ] ->
Coq <                 [(APPLIST $c1 ($LIST $lc))]
Coq < | inject_com91 [ constr9($c) ] -> [$c].
Coq < Coq < <W> Grammar extension: some rule has been masked
```

```
Coq < Check (x:nat)nat.
Toplevel input, characters 8-9
> Check (x:nat)nat.
>         ^
Syntax error: ')' expected after [Constr.constr10] (in [Constr.constr0])
```

But the factorization does not work, thus the product rule is never selected since identifiers match the `constr10` grammar. The trick is to parse the ident as a `constr10` and check *a posteriori* that the term is indeed an identifier:

```
Coq < Grammar constr constr0 : ast :=
Coq <   product [ "(" constr10($c) ":" constr($c1) ")" constr0($c2) ] ->
Coq <             [(PROD $c1 [$c]$c2)].
```

```
Coq < Check (x:nat)nat.
nat->nat
     : Set
```

We could have checked it explicitly with a `case` in the right-hand side of the rule, but the error message in the following example would not be as relevant:

```
Coq < Check (S O:nat)nat.
Toplevel input, characters 7-10
> Check (S O:nat)nat.
>        ^^^
Error: during interpretation of grammar rule product,
 This expression should be a simple identifier
```

This trick is not similar to the `SQUASH` node in which we could not detect the error while parsing. Here, the error pops out when trying to build an abstraction of `$c2` over the value of `$c`. Since it is not bound to a variable, the right-hand side of the product grammar rule fails.

9.3 Writing your own pretty printing rules

There is a mechanism for extending the vernacular's printer by adding, in the interactive toplevel, new printing rules. The printing rules are stored into a table and will be recovered at the moment of the printing by the vernacular's printer.

The user can print new constants, tactics and vernacular phrases with his desired syntax. The printing rules for new constants should be written *after* the definition of the constants. The rules should be outside a section if the user wants them to be exported.

The printing rules corresponding to the heart of the system (primitive tactics, terms and the vernacular language) are defined, respectively, in the files `PPTactic.v` and `PPConstr.v` (in the directory `syntax`). These files are automatically loaded in the initial state. The user is not expected to modify these files unless he dislikes the way primitive things are printed, in which case he will have to compile the system after doing the modifications.

When the system fails to find a suitable printing rule, a tag `#GENTERM` appears in the message.

In the following we give some examples showing how to write the printing rules for the non-terminal and terminal symbols of a grammar. We will test them frequently by inspecting the error messages. Then, we give the grammar of printing rules and a description of its semantics.

9.3.1 The Printing Rules

The printing of non terminals

The printing is the inverse process of parsing. While a grammar rule maps an input stream of characters into an AST, a printing rule maps an AST into an output stream of printing orders. So given a certain grammar rule, the printing rule is generally obtained by inverting the grammar rule.

Like grammar rules, it is possible to define several rules at the same time. The exact syntax for complex rules is described in 9.3.2. A simple printing rule is of the form:

`Syntax` *universe* `level` *precedence* **:** *name* `[` *pattern* `] -> [` *printing-orders* `]` **.**

where :

- *universe* is an identifier denoting the universe of the AST to be printed. They have the same meaning as grammar universes. The vernac universe has no equivalent in pretty-printing since vernac phrases are never printed by the system. Error messages are reported by re-displaying what the user typed in.

- *precedence* is positive integer indicating the precedence of the rule. In general the precedence for tactics is 0. The universe of terms is implicitly stratified by the hierarchy of the parsing rules. We have non terminals *constr0*, *constr1*, ..., *constr10*. The idea is that objects parsed with the non terminal $constr_i$ have precedence i. In most of the cases we fix the precedence of the printing rules for commands to be the same number of the non terminal with which it is parsed.

  A precedence may also be a triple of integers. The triples are ordered in lexicographic order, and the level n is equal to `[`n `0 0]`.

- *name* is the name of the printing rule. A rule is identified by both its universe and name, if there are two rules with both the same name and universe, then the last one overrides the former.

- *pattern* is a pattern that matches the AST to be printed. The syntax of patterns is dependent on the universe of the AST to be printed (e.g. patterns are parsed as `constr` if the universe is `constr`, etc), and a quotation can be used to escape the default parser associated to this universe. A description of the syntax of patterns is given in section 9.1.

- *printing-orders* is the sequence of orders indicating the concrete layout of the printer.

Example 1: *Syntax for user-defined tactics.*

The first usage of the `Syntax` command might be the printing order for a user-defined tactic:

```
Coq < Declare ML Module "eqdecide".

Coq < Syntax tactic level 0:
Coq <   Compare_PP [(Compare $com1 $com2)]    ->
Coq <              ["Compare" [1 2] $com1 [1 2] $com2].
```

If such a printing rule is not given, a disgraceful `#GENTERM` will appear when typing `Show Script` or `Save`. For a tactic macro defined by a `Tactic Definition` command, a printing rule is automatically generated so the user don't have to write one.

Example 2: *Defining the syntax for new constants.*

Let's define the constant `Xor` in Coq:

```
Coq < Definition Xor := [A,B:Prop] A/\~B \/ ~A/\B.
```

Given this definition, we want to use the syntax of `A X B` to denote `(Xor A B)`. To do that we give the grammar rule:

```
Coq < Grammar constr constr7 :=
Coq <   Xor [ constr6($c1) "X" constr7($c2) ] -> [(Xor $c1 $c2)].
```

Note that the operator is associative to the right. Now `True X False` is well parsed:

```
Coq < Goal True X False.
1 subgoal

  ============================
   (Xor True False)
```

To have it well printed we extend the printer:

```
Coq < Syntax constr level 7:
Coq <   Pxor  [(Xor  $t1  $t2)] -> [ $t1:L  " X "  $t2:E ].
```

and now we have the desired syntax:

```
Coq < Show.
1 subgoal

  ============================
   True X False
```

Let's comment the rule:

- `constr` is the universe of the printing rule.
- `7` is the rule's precedence and it is the same one than the parsing production (constr7).
- `Pxor` is the name of the printing rule.

- `(Xor $t1 $t2)` is the pattern of the term to be printed. Between `<< >>` we are allowed to use the syntax of arbitrary AST instead of terms. Metavariables may occur in the pattern but preceded by `$`.
- `$t1:L " X " $t2:E` are the printing orders, it tells to print the value of `$t1` then the symbol `X` and then the value of `$t2`.

  The `L` in the little box `$t1:L` indicates not to put parentheses around the value of `$t1` if its precedence is **less** than the rule's one. An `E` instead of the `L` would mean not to put parentheses around the value of `$t1` if its the precedence is **less or equal** than the rule's one.

  The associativity of the operator can be expressed in the following way:

  `$t1:L " X " $t2:E` associates the operator to the right.

  `$t1:E " X " $t2:L` associates to the left.

  `$t1:L " X " $t2:L` is non-associative.

Note that while grammar rules are related by the name of non-terminals (such as `constr6` and `constr7`) printing rules are isolated. The `Pxor` rule tells how to print an `Xor` expression but not how to print its subterms. The printer looks up recursively the rules for the values of `$t1` and `$t2`. The selection of the printing rules is strictly determined by the structure of the AST to be printed.

This could have been defined with the `Infix` command.

Example 3: *Forcing to parenthesize a new syntactic construction*

You can force to parenthesize a new syntactic construction by fixing the precedence of its printing rule to a number greater than 9. For example a possible printing rule for the `Xor` connector in the prefix notation would be:

```
Coq < Syntax constr level 10:
Coq <   ex_imp [(Xor $t1  $t2)] -> [ "X " $t1:L " " $t2:L ].
```

No explicit parentheses are contained in the rule, nevertheless, when using the connector, the parentheses are automatically written:

```
Coq < Show.
1 subgoal

   ============================
    (X True False)
```

A precedence higher than 9 ensures that the AST value will be parenthesized by default in either the empty context or if it occurs in a context where the instructions are of the form `$t:L` or `$t:E`.

Example 4: *Dealing with list patterns in the syntax rules*

The following productions extend the parser to recognize a tactic called `MyIntros` that receives a list of identifiers as argument as the primitive `Intros` tactic does:

```
Coq < Grammar tactic simple_tactic: ast :=
Coq <   my_intros [ "MyIntros" ne_identarg_list($idl) ] ->
Coq <             [(MyIntrosWith ($LIST $idl))].
```

To define the printing rule for `MyIntros` it is necessary to define the printing rule for the non terminal `ne_identarg_list`. In grammar productions the dependency between the non terminals is explicit. This is not the case for printing rules, where the dependency between the rules is determined by the structure of the pattern. So, the way to make explicit the relation between printing rules is by adding structure to the patterns.

```
Coq < Syntax tactic level 0:
Coq <   myintroswith [<<(MyIntrosWith ($LIST $L))>>] ->
Coq <    [ "MyIntros " (NEIDENTARGLIST ($LIST $L)) ].
```

This rule says to print the string `MyIntros` and then to print the value of `(NEIDENTARGLIST ($LIST $L))`.

```
Coq < Syntax tactic level 0:
Coq <   ne_identarg_list_cons [<<(NEIDENTARGLIST $id ($LIST $l))>>]
Coq <                              -> [ $id " " (NEIDENTARGLIST ($LIST $l)) ]
Coq < | ne_identarg_list_single [<<(NEIDENTARGLIST $id)>>] -> [ $id ].
```

The first rule says how to print a non-empty list, while the second one says how to print the list with exactly one element. Note that the pattern structure of the binding in the first rule ensures its use in a recursive way.

Like the order of grammar productions, the order of printing rules *does matter*. In case of two rules whose patterns superpose each other the **last** rule is always chosen. In the example, if the last two rules were written in the inverse order the printing will not work, because only the rule *ne_identarg_list_cons* would be recursively retrieved and there is no rule for the empty list. Other possibilities would have been to write a rule for the empty list instead of the *ne_identarg_list_single* rule,

```
Coq < Syntax tactic level 0:
Coq <   ne_identarg_list_nil [<<(NEIDENTARGLIST)>>] -> [ ].
```

This rule indicates to do nothing in case of the empty list. In this case there is no superposition between patterns (no critical pairs) and the order is not relevant. But a useless space would be printed after the last identifier.

Example 5: *Defining constants with arbitrary number of arguments*

Sometimes the constants we define may have an arbitrary number of arguments, the typical case are polymorphic functions. Let's consider for example the functional composition operator. The following rule extends the parser:

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

Coq < Grammar constr constr6 :=
Coq <   expl_comp [constr5($c1) "o" constr6($c2) ] ->
Coq <             [(explicit_comp ? ? ? $c1 $c2)].
```

Our first idea is to write the printing rule just by "inverting" the production:

```
Coq < Syntax constr level 6:
Coq <   expl_comp [(explicit_comp ? ? ? $f $g)] -> [ $f:L "o" $g:L ].
```

This rule is not correct: `?` is an ordinary AST (indeed, it is the AST `(XTRA "ISEVAR")`), and does not behave as the "wildcard" pattern `$_`. Here is a correct version of this rule:

```
Coq < Syntax constr level 6:
Coq <   expl_comp [(explicit_comp $_ $_ $_ $f $g)] -> [ $f:L "o" $g:L ].
```

Let's test the printing rule:

```
Coq < Definition Id := [A:Set][x:A]x.
Id is defined

Coq < Check (Id nat) o (Id nat).
(Id nat)o(Id nat)
     : nat->nat

Coq < Check ((Id nat)o(Id nat)  O).
(explicit_comp nat nat nat (Id nat) (Id nat) O)
     : nat
```

In the first case the rule was used, while in the second one the system failed to match the pattern of the rule with the AST of `((Id nat)o(Id nat) O)`. Internally the AST of this term is the same as the AST of the term `(explicit_comp nat nat nat (Id nat) (Id nat) O)`. When the system retrieves our rule it tries to match an application of six arguments with an application of five arguments (the AST of `(explicit_comp $_ $_ $_ $f $g)`). Then, the matching fails and the term is printed using the rule for application.

Note that the idea of adding a new rule for `explicit_comp` for the case of six arguments does not solve the problem, because of the polymorphism, we can always build a term with one argument more. The rules for application deal with the problem of having an arbitrary number of arguments by using list patterns. Let's see these rules:

```
Coq < Syntax constr level 10:
Coq <   app [<<(APPLIST $H ($LIST $T))>>] ->
Coq <        [ [<hov 0>  $H:E (APPTAIL ($LIST $T)):E ] ]
Coq <
Coq < | apptailcons [<<(APPTAIL $H ($LIST $T))>>] ->
Coq <        [ [1 1] $H:L (APPTAIL ($LIST $T)):E ]
Coq < | apptailnil [<<(APPTAIL)>>] -> [ ].
```

The first rule prints the operator of the application, and the second prints the list of its arguments. Then, one solution to our problem is to specialize the first rule of the application to the cases where the operator is `explicit_comp` and the list pattern has **at least** five arguments:

```
Coq < Syntax constr level 10:
Coq <   expl_comp
Coq <     [<<(APPLIST <<explicit_comp>> $_ $_ $_ $f $g ($LIST $l))>>]
Coq <      -> [ [<hov 0> $f:L "o" $g:L (APPTAIL ($LIST $l)):E ] ].
```

Now we can see that this rule works for any application of the operator:

```
Coq < Check ((Id nat) o (Id nat) O).
((Id nat)o(Id nat) O)
     : nat

Coq < Check ((Id nat->nat) o (Id nat->nat) [x:nat]x O).
((Id nat->nat)o(Id nat->nat) [x:nat]x O)
     : nat
```

In the examples presented by now, the rules have no information about how to deal with indentation, break points and spaces, the printer will write everything in the same line without spaces. To indicate the concrete layout of the patterns, there's a simple language of printing instructions that will be described in the following section.

The printing of terminals

The user is not expected to write the printing rules for terminals, this is done automatically. Primitive printing is done for identifiers, strings, paths, numbers. For example :

```
Coq < Grammar vernac vernac: ast :=
Coq <   mycd [ "MyCd" prim:string($dir) "." ] -> [(MYCD $dir)].

Coq < Syntax vernac level 0:
Coq <   mycd [<<(MYCD $dir)>>] -> [ "MyCd " $dir ].
```

There is no more need to encapsulate the `$dir` meta-variable with the `$PRIM` or the `$STR` operator as in the version 6.1. However, the pattern `(MYCD ($STR $dir))` would be safer, because the rule would not be selected to print an ill-formed AST. The name of default primitive printer is the Objective Caml function `print_token`. If the user wants a particular terminal to be printed by another printer, he may specify it in the right part of the rule. Example:

```
Coq < Syntax tactic level 0 :
Coq <   do_pp [<<(DO ($NUM $num) $tactic)>>]
Coq <      -> [ "Do " $num:"my_printer" [1 1] $tactic ].
```

The printer *my_printer* must have been installed as shown below.

Primitive printers

Writing and installing primitive pretty-printers requires to have the sources of the system like writing tactics.

A primitive pretty-printer is an Objective Caml function of type

```
Esyntax.std_printer -> CoqAst.t -> Pp.std_ppcmds
```

The first argument is the global printer, it can be called for example by the specific printer to print subterms. The second argument is the AST to print, and the result is a stream of printing orders like :

- `'sTR"`*string*`"` to print the string *string*
- `'bRK` *num1 num2* that has the same semantics than `[` *num1 num2* `]` in the print rules.
- `'sPC` to leave a blank space
- `'iNT` n to print the integer n
- ...

There is also commands to make boxes (`h` or `hv`, described in file `lib/pp.mli`). Once the printer is written, it must be registered by the command :

```
Esyntax.Ppprim.add ("name",my_printer);;
```

Then, in the toplevel, after having loaded the right Objective Caml module, it can be used in the right hand side of printing orders using the syntax `$truc:"name"`.

The real name and the registered name of a pretty-printer does not need to be the same. However, it can be nice and simple to give the same name.

9.3.2 Syntax for pretty printing rules

This section describes the syntax for printing rules. The metalanguage conventions are the same as those specified for the definition of the *pattern*'s syntax in section 9.1. The grammar of printing rules is the following:

<table>
<tr><td>printing-rule</td><td>::=</td><td><code>Syntax</code> ident level <code>;</code> ... <code>;</code> level</td></tr>
<tr><td>level</td><td>::=</td><td><code>level</code> precedence <code>:</code> rule | ... | rule</td></tr>
<tr><td>precedence</td><td>::=</td><td>integer | <code>[</code> integer integer integer <code>]</code></td></tr>
<tr><td>rule</td><td>::=</td><td>ident <code>[</code> pattern <code>]</code> <code>-></code> <code>[</code> [printing-order ... printing-order] <code>]</code></td></tr>
<tr><td>printing-order</td><td>::=</td><td><code>FNL</code></td></tr>
<tr><td></td><td>|</td><td>string</td></tr>
<tr><td></td><td>|</td><td><code>[</code> integer integer <code>]</code></td></tr>
<tr><td></td><td>|</td><td><code>[</code> box [printing-order ... printing-order] <code>]</code></td></tr>
<tr><td></td><td>|</td><td>ast [<code>:</code> prim-printer] [<code>:</code> paren-rel]</td></tr>
<tr><td>box</td><td>::=</td><td>< box-type integer ></td></tr>
<tr><td>box-type</td><td>::=</td><td><code>hov</code> | <code>hv</code> | <code>v</code> | <code>h</code></td></tr>
<tr><td>paren-rel</td><td>::=</td><td><code>L</code> | <code>E</code></td></tr>
<tr><td>prim-printer</td><td>::=</td><td>string</td></tr>
<tr><td>pattern</td><td>::=</td><td>ast-quot | << ast >></td></tr>
</table>

Non-terminal *ast-quot* is the default quotation associated to the extended universe. Patterns not belonging to the input syntax can be given directly as AST using << >>.

As already stated, the order of rules in a given level is relevant (the last ones override the previous ones).

Pretty grammar structures

The basic structure is the printing order sequence. Each order has a printing effect and they are sequentially executed. The orders can be:

- printing orders
- printing boxes

Printing orders Printing orders can be of the form:

- `"`*string*`"` prints the *string*.
- `FNL` force a new line.

- `$t`:*paren-rel* or `$t`:*prim-printer*:*paren-rel*

  ast is used to build an AST in current context. The printer looks up the adequate printing rule and applies recursively this method. The optional field *prim-printer* is a string with the name primitive pretty-printer to call (The name is not the name of the Objective Caml function, but the name given to `Esyntax.Ppprim.add`). Recursion of the printing is determined by the pattern's structure. *paren-rel* is the following:

  `L` if t 's precedence is **less** than the rule's one, then no parentheses around t are written.
  `E` if t 's precedence is **less or equal** than the rule's one then no parentheses around t are written.
  none **never** write parentheses around t.

Printing boxes The concept of formatting boxes is used to describe the concrete layout of patterns: a box may contain many objects which are orders or subboxes sequences separated by breakpoints; the box wraps around them an imaginary rectangle.

1. **Box types**

   The type of boxes specifies the way the components of the box will be displayed and may be:

   - `h` : to catenate objects horizontally.
   - `v` : to catenate objects vertically.
   - `hv` : to catenate objects as with an "h box" but an automatic vertical folding is applied when the horizontal composition does not fit into the width of the associated output device.
   - `hov` : to catenate objects horizontally but if the horizontal composition does not fit, a vertical composition will be applied, trying to catenate horizontally as many objects as possible.

   The type of the box can be followed by a n offset value, which is the offset added to the current indentation when breaking lines inside the box.

2. **Boxes syntax**

   A box is described by a sequence surrounded by `[ ]`. The first element of the sequence is the box type: this type surrounded by the symbols < > is one of the words `hov`, `hv`, `v`, `v` followed by an offset. The default offset is 0 and the default box type is `h`.

3. **Breakpoints**

   In order to specify where the pretty-printer is allowed to break, one of the following breakpoints may be used:

   - `[0 0]` is a simple break-point, if the line is not broken here, no space is included ("Cut").
   - `[1 0]` if the line is not broken then a space is printed ("Spc").

- `[i j]` if the line is broken, the value j is added to the current indentation for the following line; otherwise i blank spaces are inserted ("Brk").

Examples : It is interesting to test printing rules on "small" and "large" expressions in order to see how the break of lines and indentation are managed. Let's define two constants and make a `Print` of them to test the rules. Here are some examples of rules for our constant `Xor`:

```
Coq < Definition A := True X True.

Coq < Definition B := True X True X True X True X True X True X True
Coq <                  X True X True X True X True X True X True.

Coq < Syntax constr level 6:
Coq <   Pxor [(Xor $t1 $t2)] -> [ $t1:L " X " $t2:E ].
```

This rule prints everything in the same line exceeding the line's width.

```
Coq < Print B.
B =
True X True X True X True X True X True X True X True X True X True X Tru
e X True X True
     : Prop
```

Let's add some break-points in order to force the printer to break the line before the operator:

```
Coq < Syntax constr level 6:
Coq <   Pxor [(Xor $t1 $t2)] -> [ $t1:L [0 1] " X " $t2:E ].

Coq < Print B.
B = True X True X True X True X True X True X True X True X True X True
  X True X True X True
     : Prop
```

The line was correctly broken but there is no indentation at all. To deal with indentation we use a printing box:

```
Coq < Syntax constr level 6:
Coq <   Pxor [(Xor $t1 $t2)] ->
Coq <    [ [<hov 0> $t1:L [0 1] " X " $t2:E ] ].
```

With this rule the printing of A is correct, an the printing of B is indented.

```
Coq < Print B.
B =
True
  X True
      X True
          X True
              X True
                  X True
                      X True X True X True X True X True X True X True
     : Prop
```

If we had chosen the mode `v` instead of `hov` :

```
Coq < Syntax constr level 6:
Coq <   Pxor [(Xor $t1 $t2)] -> [ [<v 0> $t1:L [0 1] " X " $t2:E ] ].
```

We would have obtained a vertical presentation:

```
Coq < Print A.
A = True
      X True
     : Prop
```

The difference between the presentation obtained with the `hv` and `hov` type box is not evident at first glance. Just for clarification purposes let's compare the result of this silly rule using an `hv` and a `hov` box type:

```
Coq < Syntax constr level 6:
Coq <   Pxor [(Xor $t1 $t2)] ->
Coq <    [ [<hv 0> "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
Coq <        [0 0] "---------------------------"
Coq <        [0 0] "ZZZZZZZZZZZZZZZZ" ] ].

Coq < Print A.
A =
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
---------------------------
ZZZZZZZZZZZZZZZZ
     : Prop

Coq < Syntax constr level 6:
Coq <   Pxor [(Xor $t1 $t2)] ->
Coq <    [ [<hov 0> "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
Coq <        [0 0]  "---------------------------"
Coq <        [0 0]  "ZZZZZZZZZZZZZZZZ" ] ].

Coq < Print A.
A =
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
---------------------------ZZZZZZZZZZZZZZZZ
     : Prop
```

In the first case, as the three strings to be printed do not fit in the line's width, a vertical presentation is applied. In the second case, a vertical presentation is applied, but as the last two strings fit in the line's width, they are printed in the same line.

9.3.3 Debugging the printing rules

By now, there is almost no semantic check of printing rules in the system. To find out where the problem is, there are two possibilities: to analyze the rules looking for the most common errors or to work in the toplevel tracing the ml code of the printer. When the system can't find the proper rule to print an Ast, it prints `#GENTERM` *ast*. If you added no printing rule, it's probably a bug and you can send it to the Coq team.

Most common errors

Here are some considerations that may help to get rid of simple errors:

- make sure that the rule you want to use is not defined in previously closed section.
- make sure that **all** non-terminals of your grammar have their corresponding printing rules.
- make sure that the set of printing rules for a certain non terminal covers all the space of AST values for that non terminal.
- the order of the rules is important. If there are two rules whose patterns superpose (they have common instances) then it is always the most recent rule that will be retrieved.
- if there are two rules with the same name and universe the last one overrides the first one. The system always warns you about redefinition of rules.

Tracing the Objective Caml code of the printer

Some of the conditions presented above are not easy to verify when dealing with many rules. In that case tracing the code helps to understand what is happening. The printers are in the file `src/typing/printer`. There you will find the functions:

- `prterm` : the printer of constructions
- `gentacpr` : the printer of tactics

These printers are defined in terms of a general printer `genprint` (this function is located in `src/parsing/esyntax.ml`) and by instantiating it with the adequate parameters. `genprint` waits for: the universe to which this AST belongs (*tactic*, *constr*), a default printer, the precedence of the AST inherited from the caller rule and the AST to print. `genprint` looks for a rule whose pattern matches the AST, and executes in order the printing orders associated to this rule. Subterms are printed by recursively calling the generic printer. If no rule matches the AST, the default printer is used.

An AST of a universe may have subterms that belong to another universe. For instance, let v be the AST of the tactic expression `MyTactic O`. The function `gentacpr` is called to print v. This function instantiates the general printer `genprint` with the universe *tactic*. Note that v has a subterm c corresponding to the AST of `O` (c belongs to the universe *constr*). `genprint` will try recursively to print all subterms of v as belonging to the same universe of v. If this is not possible, because the subterm belongs to another universe, then the default printer that was given as argument to `genprint` is applied. The default printer is responsible for changing the universe in a proper way calling the suitable printer for c.

Technical Remark. In the file `PPTactic.v`, there are some rules that do not arise from the inversion of a parsing rule. They are strongly related to the way the printing is implemented.

```
Coq < Syntax tactic level 8:
Coq <     tactic_to_constr [<<(COMMAND $c)>>] -> [ $c:"constr":9 ]
```

As an AST of tactic may have subterms that are commands, these rules allow the printer of tactic to change the universe. The primitive printer `command` is a special identifier used for this purpose. They are used in the code of the default printer that `gentacpr` gives to `genprint`.

Chapter 10

The tactic language

This chapter gives a compact documentation of the tactic language available in the toplevel of Coq. We start by giving the syntax and, next, we present the informal semantic. Finally, we show some examples which deal with small but also with non-trivial problems. If you want to know more regarding this language and especially about its fundations, you can refer to [31].

10.1 Syntax

The syntax of the tactic language is given in table 10.1. We use a BNF-like notation. Terminal symbols are set in sans serif font (like this). Non-terminal symbols are set in italic font ($like\ that$). ... | ... denotes the or operation. ...* denotes zero, one or several repetitions. ...$^+$ denotes one or several repetitions. Parentheses *(...)* denote grouping. The main entry is $expr$ and the entries nat, $ident$, $term$ and $primitive\_tactic$ represent respectively the natural numbers, the authorized identificators, Coq's terms and all the basic tactics. In $term$, there can be specific variables like ?n where n is a nat or ?, which are metavariables for pattern matching. ?n allows us to keep instantiations and to make constraints whereas ? shows that we are not interested in what will be matched.

This language is used in proof mode but it can also be used in toplevel definitions as shown in table 10.2.

10.2 Semantic

10.2.1 Values

Values are given by table 10.3. All these values are tactic values, i.e. to be applied to a goal, except Fun, Rec and arg values.

10.2.2 Evaluation

Local definitions

Local definitions can be done as follows:

```
expr          ::= expr ; expr
                | expr ; [ (expr |)* expr ]
                | atom

atom          ::= Fun input_fun+ -> expr
                | Let (let_clause And)* let_clause In expr
                | Rec rec_clause
                | Rec (rec_clause And)* rec_clause In expr
                | Match Context With (context_rule |)* context_rule
                | Match term With (match_rule |)* match_rule
                | ( expr )
                | ( expr expr+ )
                | atom Orelse atom
                | Do (int | ident) atom
                | Repeat atom
                | Try atom
                | First [ (expr |)* expr ]
                | Solve [ (expr |)* expr ]
                | Idtac
                | Fail
                | primitive_tactic
                | arg

input_fun     ::= ident
                | ()

let_clause    ::= ident = expr

rec_clause    ::= ident input_fun+ -> expr

context_rule  ::= [ (context_hyps ;)* context_hyps |- term ] -> expr
                | [ |- term ] -> expr
                | _ -> expr

context_hyps  ::= ident : term
                | _ : term

match_rule    ::= [ term ] -> expr
                | _ -> expr

arg           ::= ()
                | nat
                | ident
                | 'term
```

Table 10.1: Syntax of the tactic language

top	::=	Tactic Definition $ident$ $input\_fun^*$:= $expr$
	|	Recursive Tactic Definition ($trec\_clause$ And)* $trec\_clause$
$trec\_clause$	::=	$ident$ $input\_fun^+$:= $expr$

Table 10.2: Tactic toplevel definitions

$vexpr$	::=	$vexpr$; $vexpr$
	|	$vexpr$; [($vexpr$ |)* $vexpr$]
	|	$vatom$
$vatom$	::=	Fun $input\_fun^+$ -> $expr$
	|	Rec rec_clause
	|	Rec (rec_clause And)* rec_clause In $expr$
	|	Match Context With ($context\_rule$ |)* $context\_rule$
	|	($vexpr$)
	|	$vatom$ Orelse $vatom$
	|	Do (int | $ident$) $vatom$
	|	Repeat $vatom$
	|	Try $vatom$
	|	First [($vexpr$ |)* $vexpr$]
	|	Solve [($vexpr$ |)* $vexpr$]
	|	Idtac
	|	Fail
	|	$primitive\_tactic$
	|	arg

Table 10.3: Values of $\mathcal{L}_{tac}$

Let $ident_1$ = $expr_1$
And $ident_2$ = $expr_2$
...
And $ident_n$ = $expr_n$ In
$expr$

$expr_i$ is evaluated to v_i, then, $expr$ is evaluated by subsituting v_i to each occurrence of $ident_i$, for $i = 1, ..., n$. There is no dependencies between the $expr_i$ and the $ident_i$.

Pattern matching on terms

We can carry out pattern matching on terms with:

Match $term$ With
  $term_1$ -> $expr_1$
| $term_2$ -> $expr_2$
...
| $term_n$ -> $expr_n$
| _ -> $expr_{n+1}$

if $term$ is matched (non-linear first order unification) by $term_1$ then $expr_1$ is evaluated by substituting the pattern matching instantiations to the metavariables. Else, $term_2$ is tried and so on. If no $term_i$, with $i = 1, ..., n$, matches $term$ then the last clause is used and $expr_{n+1}$ is evaluated.

Error message:

`No matching clauses for Match`
No pattern can be used and, in particular, there is no _ pattern.

Application

An application is an expression of the following form:

($expr$ $expr_1$ ... $expr_n$)

$expr$ is evaluated to v and $expr_i$ is evaluated to v_i, for $i = 1, ..., n$. If $expr$ is a Fun or Rec value then the body is evaluated by substituting v_i to the formal parameters, for $i = 1, ..., n$. For recursive clauses, the bodies are lazily substituted (when an identifier to be evaluated is the name of a recursive clause).

10.2.3 Application of tactic values

Sequence

A sequence is an expression of the following form:

$expr_1$; $expr_2$

$expr_1$ and $expr_2$ are evaluated to v_1 and v_2. v_1 and v_2 must be tactic values. v_1 is then applied and v_2 is applied to the subgoals generated by the application of v_1. Sequence is left associating.

General sequence

We can generalize the previous sequence operator by:

$expr_0$; [$expr_1$ | ... | $expr_n$]

$expr_i$ is evaluated to v_i, for $i = 0, ..., n$. v_0 is applied and v_i is applied to the i-th generated subgoal by the application of v_0, for $= 1, ..., n$. It fails if the application of v_0 does not generate exactly n subgoals.

Branching

We can easily branch with the following structure:

$expr_1$ Orelse $expr_2$

$expr_1$ and $expr_2$ are evaluated to v_1 and v_2. v_1 and v_2 must be tactic values. v_1 is applied and if it fails then v_2 is applied. Branching is left associating.

For loop

We have a for loop with:

Do n $expr$

$expr$ is evaluated to v. v must be a tactic value. v is applied n times. Supposing $n > 1$, 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 n applications have been completed.

Repeat loop

We have a repeat loop with:

Repeat $expr$

$expr$ is evaluated to v. v must be a tactic value. v is applied until it fails. Supposing $n > 1$, after the first application of v, v is applied, at least once, to the generated subgoals and so on. It stops when it fails for all the generated subgoals. It never fails.

Error catching

We can catch the tactic errors with:

Try $expr$

$expr$ is evaluated to v. v must be a tactic value. v is applied. If the application of v fails, it catches the error and leaves the goal unchanged. It never fails.

First tactic to work

We may consider the first tactic to work (i.e. which does not fail) among a panel of tactics:

First [$expr_1$ | ... | $expr_n$]

$expr_i$ are evaluated to v_i and v_i must be tactic values, for $i = 1, ..., n$. Supposing $n > 1$, it applies v_1, if it works, it stops else it tries to apply v_2 and so on. It fails when there is no applicable tactic.

Error message:

```
No applicable tactic
```

Solving

We may consider the first to solve (i.e. which generates no subgoal) among a panel of tactics:

Solve [$expr_1$ | ... | $expr_n$]

$expr_i$ are evaluated to v_i and v_i must be tactic values, for $i = 1, ..., n$. Supposing $n > 1$, it applies v_1, if it solves, it stops else it tries to apply v_2 and so on. It fails if there is no solving tactic.

Error message:

```
Cannot solve the goal
```

Identity

We have the identity tactic:

Idtac

It leaves the goal unchanged but it appears in the proof script.

Failing

We have the failing tactic:

Fail

It always fails and leaves the goal unchanged. It does not appear in the proof script and can be catched by Try.

Error message:

Fail tactic always fails (level n).

Pattern matching on proof contexts

We can make pattern matching on proof contexts using the following expression:

Match Context With
 [$context\_hyps_{1,1}$;...;$context\_hyps_{1,m_1}$ |-$term_1$] -> $expr_1$
 |[$context\_hyps_{2,1}$;...;$context\_hyps_{2,m_2}$ |-$term_2$] -> $expr_2$
 ...
 |[$context\_hyps_{n,1}$;...;$context\_hyps_{n,m_n}$ |-$term_n$] -> $expr_n$
 |_ -> $expr_{n+1}$

If each hypothesis pattern $context\_hyps_{1,i}$, with $i = 1, ..., m_1$ is matched (non-linear first order unification) by an hypothesis of the goal and if $term_1$ is matched by the conclusion of the goal, then $expr_1$ is evaluated to v_1 by substituting the pattern matching to the metavariables and the real hypothesis names bound to the possible hypothesis names occurring in the hypothesis patterns. If v_1 is a tactic value, then it is applied to the goal. If this application fails, then another combination of hypotheses is tried with the same proof context pattern. If there is no other combination of hypotheses then the second proof context pattern is tried and so on. If the next to last proof context pattern fails then $expr_{n+1}$ is evaluated to v_{n+1} and v_{n+1} is applied.

Error message:

`No matching clauses for Match Context`
No proof context pattern can be used and, in particular, there is no _ proof context pattern.

10.2.4 Tactic toplevel definitions

Basically, tactics toplevel definitions are made as follows:

Tactic Definition $ident$:= $expr$

$expr$ is evaluated to v and v is associated to $ident$. Next, every script is evaluated by substituting v to $ident$.

We can define functional definitions by:

Tactic Definition $ident$ $input\_fun_1$ ... $input\_fun_n$:= $expr$

This definition is nothing else than syntactical sugar for:

Tactic Definition $ident$:= Fun $input\_fun_1$ ... $input\_fun_n$ -> $expr$

Then, this definition is treated as above.

Finally, mutal recursive function definitions are possible with:

Recursive Tactic Definition
$ident_1$ $input\_fun_{1,1}$ ... $input\_fun_{1,m_1}$:= $expr_1$
And $ident_2$ $input\_fun_{2,1}$ ... $input\_fun_{2,m_2}$:= $expr_2$
...
And $ident_n$ $input\_fun_{n,1}$ ... $input\_fun_{n,m_n}$:= $expr_n$

This definition bloc is a set of simultaneous functional definitions (use of the same previous syntactical sugar) and the other scripts are evaluated as usual except that the substitutions are lazily carried out (when an identifier to be evaluated is the name of a recursive definition).

10.3 Examples

10.3.1 About the cardinality of the natural number set

A first example which shows how to use the pattern matching over the proof contexts is the proof that natural numbers have more than two elements. The proof of such a lemma can be done as shown in table 10.4.

```
Coq < Lemma card_nat: ~(EX x:nat|(EX y:nat|(z:nat)(x=z)/\(y=z))).

Coq < Proof.

Coq < Red;Intro H.

Coq < Elim H;Intros a Ha.

Coq < Elim Ha;Intros b Hb.

Coq < Elim (Hb (0));Elim (Hb (1));Elim (Hb (2));Intros;
Coq <   Match Context With
Coq <     [_:?1=?2;_:?1=?3|-?] ->
Coq <       Cut ?2=?3;[Discriminate|Apply trans_equal with ?1;Auto].

Coq < Save.
```

Table 10.4: A proof on cardinality of natural numbers

We can notice that all the (very similar) cases coming from the three eliminations (with three distinct natural numbers) are successfully solved by a Match Context structure and, in particular, with only one pattern (use of non-linear unification).

10.3.2 Permutation on closed lists

Another more complex example is the problem of permutation on closed lists. The aim is to show that a closed list is a permutation of another one.

First, we define the permutation predicate as shown in table 10.5.

```
Coq < Section Sort.

Coq < Variable A:Set.

Coq <
Coq < Inductive permut:(list A)->(list A)->Prop:=
Coq <   permut_refl:(l:(list A))(permut l l)
Coq <  |permut_cons:
Coq <     (a:A)(l0,l1:(list A))(permut l0 l1)->
Coq <       (permut (cons a l0) (cons a l1))
Coq <  |permut_append:
Coq <     (a:A)(l:(list A))(permut (cons a l) (l^(cons a (nil A))))
Coq <  |permut_trans:
Coq <     (l0,l1,l2:(list A))(permut l0 l1)->(permut l1 l2)->
Coq <       (permut l0 l2).

Coq < End Sort.
```

Table 10.5: Definition of the permutation predicate

Next, we can write naturally the tactic and the result can be seen in table 10.6. We can notice that we use two toplevel definitions PermutProve and Permut. The function to be called is PermutProve which computes the lengths of the two lists and calls Permut with the length if the two lists have the same length. Permut works as expected. If the two lists are equal, it concludes. Otherwise, if the lists have identical first elements, it applies Permut on the tail of the lists. Finally, if the lists have different first elements, it puts the first element of one of the lists (here the second one which appears in the permut predicate) at the end if that is possible, i.e., if the new first element has been at this place previously. To verify that all rotations have been done for a list, we use the length of the list as an argument for Permut and this length is decremented for each rotation down to, but not including, 1 because for a list of length n, we can make exactly $n - 1$ rotations to generate at most n distinct lists. Here, it must be noticed that we use the natural numbers of Coq for the rotation counter. In table 10.1, we can see that it is possible to use usual natural numbers but they are only used as arguments for primitive tactics and they cannot be handled, in particular, we cannot make computations with them. So, a natural choice is to use Coq data structures so that Coq makes the computations (reductions) by Eval Compute in and we can get the terms back by Match.

With PermutProve, we can now prove lemmas such those shown in table 10.7.

10.3.3 Deciding intuitionistic propositional logic

The pattern matching on proof contexts allows a complete and so a powerful backtracking when returning tactic values. An interesting application is the problem of deciding intuitionistic propositional logic. Considering the contraction-free sequent calculi LJT* of Roy Dyckhoff ([44]), it is quite natural to code such a tactic using the tactic language as shown in table 10.8. The tactic Axioms tries to conclude using usual axioms. The tactic Simplif applies all the reversible rules of Dyckhoff's system. Finally, the tactic TautoProp (the main tactic to be called) simplifies with Simplif, tries to conclude with Axioms and tries several paths using the backtracking rules (one of the four Dyckhoff's rules for the left implication to get rid of the contraction and the right or).

For example, with TautoProp, we can prove tautologies like those in table 10.9.

10.3.4 Deciding type isomorphisms

A more tricky problem is to decide equalities between types and modulo isomorphisms. Here, we choose to use the isomorphisms of the simply typed λ-calculus with Cartesian product and $unit$ type (see, for example, [33]). The axioms of this λ-calculus are given by table 10.10.

The tactic to judge equalities modulo this axiomatization can be written as shown in tables 10.11 and 10.12. The algorithm is quite simple. Types are reduced using axioms that can be oriented (this done by MainSimplif). The normal forms are sequences of Cartesian products without Cartesian product in the left component. These normal forms are then compared modulo permutation of the components (this is done by CompareStruct). The main tactic to be called and realizing this algorithm is IsoProve.

Table 10.13 gives examples of what can be solved by IsoProve.

```
Coq < Tactic Definition Permut n:=
Coq <  Match Context With
Coq <    [|-(permut ? ?1 ?1)] -> Apply permut_refl
Coq <   |[|-(permut ? (cons ?1 ?2) (cons ?1 ?3))] ->
Coq <      Let newn=Eval Compute in (length ?2) In
Coq <      Apply permut_cons;(Permut newn)
Coq <   |[|-(permut ?1 (cons ?2 ?3) ?4)] ->
Coq <      (Match Eval Compute in n With
Coq <         [(1)] -> Fail
Coq <        |_ ->
Coq <          Let l0'='(?3^(cons ?2 (nil ?1))) In
Coq <          Apply (permut_trans ?1 (cons ?2 ?3) l0' ?4);
Coq <          [Apply permut_append|
Coq <           Compute;(Permut '(pred n))]).
Permut is defined

Coq <
Coq < Tactic Definition PermutProve:=
Coq <  Match Context With
Coq <    [|-(permut ? ?1 ?2)] ->
Coq <      (Match Eval Compute in ((length ?1)=(length ?2)) With
Coq <         [?1=?1] -> (Permut ?1)).
PermutProve is defined
```

Table 10.6: Permutation tactic

```
Coq < Lemma permut_ex1:
Coq <   (permut nat (cons (1) (cons (2) (cons (3) (nil nat))))
Coq <      (cons (3) (cons (2) (cons (1) (nil nat))))).

Coq < Proof.

Coq < PermutProve.

Coq < Save.

Coq <
Coq < Lemma permut_ex2:
Coq <  (permut nat
Coq <    (cons (0) (cons (1) (cons (2) (cons (3) (cons (4) (cons (5)
Coq <      (cons (6) (cons (7) (cons (8) (cons (9) (nil nat)))))))))))
Coq <    (cons (0) (cons (2) (cons (4) (cons (6) (cons (8) (cons (9)
Coq <      (cons (7) (cons (5) (cons (3) (cons (1) (nil nat)))))))))))).

Coq < Proof.

Coq < PermutProve.

Coq < Save.
```

Table 10.7: Examples of PermutProve use

```
Coq < Tactic Definition Axioms:=
Coq <   Match Context With
Coq <    [|-True] -> Trivial
Coq <   |[_:False|- ?] -> ElimType False;Assumption
Coq <   |[_:?1|-?1] -> Auto.
Axioms is defined

Coq <
Coq < Tactic Definition Simplif:=
Coq <  Repeat
Coq <   (Intros;
Coq <    (Match Context With
Coq <      [id:~?|-?] -> Red in id
Coq <     |[id:?/\?|-?] -> Elim id;Do 2 Intro;Clear id
Coq <     |[id:?\/?|-?] -> Elim id;Intro;Clear id
Coq <     |[id:?1/\?2->?3|-?] ->
Coq <        Cut ?1->?2->?3;[Intro|Intros;Apply id;Split;Assumption]
Coq <     |[id:?1\/?2->?3|-?] ->
Coq <        Cut ?2->?3;[Cut ?1->?3;[Intros|
Coq <                                Intro;Apply id;Left;Assumption]|
Coq <                    Intro;Apply id;Right;Assumption]
Coq <     |[id0:?1->?2;id1:?1|-?] ->
Coq <        Cut ?2;[Intro;Clear id0|Apply id0;Assumption]
Coq <     |[|-?/\?] -> Split
Coq <     |[|-~?] -> Red)).
Simplif is defined

Coq <
Coq < Recursive Tactic Definition TautoProp:=
Coq <   Simplif;
Coq <    Axioms
Coq <    Orelse
Coq <    Match Context With
Coq <     [id:(?1->?2)->?3|-?] ->
Coq <       Cut ?2->?3;[Intro;Cut ?1->?2;[Intro;Cut ?3;[Intro;Clear id|
Coq <                               Apply id;Assumption]|Clear id]|
Coq <                   Intro;Apply id;Intro;Assumption];TautoProp
Coq <    |[id:~?1->?2|-?]->
Coq <       Cut False->?2;
Coq <       [Intro;Cut ?1->False;[Intro;Cut ?2;[Intro;Clear id|
Coq <                                Apply id;Assumption]|Clear id]|
Coq <        Intro;Apply id;Red;Intro;Assumption];TautoProp
Coq <    |[|-?\/?] ->
Coq <       (Left;TautoProp) Orelse (Right;TautoProp).
TautoProp is defined
```

Table 10.8: Deciding intuitionistic propositions

```
Coq < Lemma tauto_ex1:(A,B:Prop)A/\B->A\/B.
Coq < Proof.
Coq < TautoProp.
Coq < Save.
Coq <
Coq < Lemma tauto_ex2:(A,B:Prop)(~~B->B)->(A->B)->~~A->B.
Coq < Proof.
Coq < TautoProp.
Coq < Save.
```

Table 10.9: Proofs of tautologies with TautoProp

```
Coq < Section Iso_axioms.
Coq <
Coq < Variable A,B,C:Set.
Coq <
Coq < Axiom Com:(A*B)==(B*A).
Coq < Axiom Ass:(A*(B*C))==((A*B)*C).
Coq < Axiom Cur:((A*B)->C)==(A->B->C).
Coq < Axiom Dis:(A->(B*C))==((A->B)*(A->C)).
Coq < Axiom P_unit:(A*unit)==A.
Coq < Axiom AR_unit:(A->unit)==unit.
Coq < Axiom AL_unit:(unit->A)==A.
Coq <
Coq < Lemma Cons:B==C->(A*B)==(A*C).
Coq < Proof.
Coq < Intro Heq;Rewrite Heq;Apply refl_eqT.
Coq < Save.
Coq <
Coq < End Iso_axioms.
```

Table 10.10: Type isomorphism axioms

```
Coq < Recursive Tactic Definition Simplif trm:=
Coq <   Match trm With
Coq <    [(?1*?2)*?3] -> Rewrite <- (Ass ?1 ?2 ?3);Try MainSimplif
Coq <   |[(?1*?2)->?3] -> Rewrite (Cur ?1 ?2 ?3);Try MainSimplif
Coq <   |[?1->(?2*?3)] -> Rewrite (Dis ?1 ?2 ?3);Try MainSimplif
Coq <   |[?1*unit] -> Rewrite (P_unit ?1);Try MainSimplif
Coq <   |[unit*?1] -> Rewrite (Com unit ?1);Try MainSimplif
Coq <   |[?1->unit] -> Rewrite (AR_unit ?1);Try MainSimplif
Coq <   |[unit-> ?1] -> Rewrite (AL_unit ?1);Try MainSimplif
Coq <   |[?1*?2] ->
Coq <      ((Simplif ?1);Try MainSimplif) Orelse
Coq <      ((Simplif ?2);Try MainSimplif)
Coq <   |[?1-> ?2] ->
Coq <      ((Simplif ?1);Try MainSimplif) Orelse
Coq <      ((Simplif ?2);Try MainSimplif)
Coq < And MainSimplif:=
Coq <   Match Context With
Coq <    [|- ?1== ?2] -> Try (Simplif ?1);Try (Simplif ?2).
MainSimplif is defined
Simplif is defined

Coq <
Coq < Tactic Definition Length trm:=
Coq <   Match trm With
Coq <    [?*?1] ->
Coq <      Let succ=(Length ?1) In
Coq <      '(S succ)
Coq <   |_ -> '(1).
Length is defined

Coq <
Coq < Tactic Definition Assoc:= Repeat Rewrite <- Ass.
Assoc is defined
```

Table 10.11: Type isomorphism tactic (1)

```
Coq < Recursive Tactic Definition DoCompare n:=
Coq <   Match Context With
Coq <    [|-?1==?1] -> Apply refl_eqT
Coq <   |[|-(?1*?2)==(?1*?3)] ->
Coq <      Apply Cons;
Coq <      Let newn=(Length ?2) In
Coq <      (DoCompare newn)
Coq <   |[|-(?1*?2)==?3] ->
Coq <      (Match Eval Compute in n With
Coq <        [(1)] -> Fail
Coq <       |_ ->
Coq <         Pattern 1 (?1*?2);Rewrite Com;Assoc;
Coq <         (DoCompare '(pred n))).
DoCompare is defined

Coq <
Coq < Tactic Definition CompareStruct:=
Coq <   Match Context With
Coq <    [|-?1==?2] ->
Coq <      Let l1=(Length ?1)
Coq <      And l2=(Length ?2) In
Coq <      (Match Eval Compute in l1=l2 With
Coq <         [?1=?1] -> (DoCompare ?1)).
CompareStruct is defined

Coq <
Coq < Tactic Definition IsoProve:=MainSimplif;CompareStruct.
IsoProve is defined
```

Table 10.12: Type isomorphism tactic (2)

```
Coq < Lemma isos_ex1:(A,B:Set)((A*unit)*B)==(B*(unit*A)).

Coq < Proof.

Coq < Intros;IsoProve.

Coq < Save.

Coq <
Coq < Lemma isos_ex2:(A,B,C:Set)
Coq <   ((A*unit)->(B*C*unit))==
Coq <   (((A*unit)->((C->unit)*C))*(unit->(A->B))).

Coq < Proof.

Coq < Intros;IsoProve.

Coq < Save.
```

Table 10.13: Type equalities solved by IsoProve

Part IV

Practical tools

Chapter 11

The Coq commands

There are two Coq commands:

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

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

11.1 Interactive use (`coqtop`)

In the interactive mode, also known as the Coq toplevel, the user can develop his theories and proofs step by step. The Coq toplevel is run by the command `coqtop`.

They are two different binary images of Coq: the byte-code one and the native-code one (if Objective Caml provides a native-code compiler for your platform, which is supposed in the following). When invoking `coqtop` or `coqc`, the native-code version of the system is used. The command-line options `-byte` and `-opt` explicitly select the byte-code and the native-code versions, respectively.

The byte-code toplevel is based on a Caml toplevel (to allow the dynamic link of tactics). You can switch to the Caml toplevel with the command `Drop.`, and come back to the Coq toplevel with the command `Toplevel.loop();;`.

11.2 Batch compilation (`coqc`)

The `coqc` command takes a name *file* as argument. Then it looks for a vernacular file named *file*`.v`, and tries to compile it into a *file*`.vo` file (See 5.4).

Warning: The name *file* must be a regular Coq identifier, as defined in the section 1.1. It must only contain letters, digits or underscores (_). Thus it can be `/bar/foo/toto.v` but cannot be `/bar/foo/to-to.v`.

Notice that the `-byte` and `-opt` options are still available with `coqc` and allow you to select the byte-code or native-code versions of the system.

11.3 Resource file

When Coq is launched, with either `coqtop` or `coqc`, the resource file `$HOME/.coqrc.7.0` is loaded, where `$HOME` is the home directory of the user. If this file is not found, then the file `$HOME/.coqrc` is searched. You can also specify an arbitrary name for the resource file (see option `-init-file` below), or the name of another user to load the resource file of someone else (see option `-user`).

This file may contain, for instance, `Add LoadPath` commands to add directories to the load path of Coq. It is possible to skip the loading of the resource file with the option `-q`.

11.4 Environment variables

There are three environment variables used by the Coq system. `$COQBIN` for the directory where the binaries are, `$COQLIB` for the directory whrer the standard library is, and `$COQTOP` for the directory of the sources. The latter is useful only for developers that are writing their own tactics and are using `coq_makefile` (see 12.3). If `$COQBIN` or `$COQLIB` are not defined, Coq will use the default values (defined at installation time). So these variables are useful only if you move the Coq binaries and library after installation.

11.5 Options

The following command-line options are recognized by the commands `coqc` and `coqtop`:

`-byte`
: Run the byte-code version of Coq.

`-opt`
: Run the native-code version of Coq.

`-I` ***directory***, `-include` ***directory***
: Add *directory* to the searched directories when looking for a file.

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

`-is` ***file***, `-inputstate` ***file***
: Cause Coq to use the state put in the file *file* as its input state. The default state is *initial.coq*. Mainly useful to build the standard input state.

`-nois`
: Cause Coq to begin with an empty state. Mainly useful to build the standard input state.

`-notactics`
: Forbid the dynamic loading of tactics.

`-init-file` ***file***
: Take *file* as the resource file.

`-q`
Cause Coq not to load the resource file.

`-user` ***username***
Take resource file of user *username* (that is ~*username*`/.coqrc.7.0`) instead of yours.

`-load-ml-source` ***file***
Load the Caml source file *file*.

`-load-ml-object` ***file***
Load the Caml object file *file*.

`-load-vernac-source` ***file***
Load Coq file *file*`.v`

`-load-vernac-object` ***file***
Load Coq compiled file *file*`.vo`

`-require` ***file***
Load Coq compiled file *file*`.vo` and import it (`Require` *file*).

`-compile` ***file***
This compiles file *file*`.v` into *file*`.vo`. This option implies options `-batch` and `-silent`. It is only available for `coqtop`.

`-batch`
Batch mode : exit just after arguments parsing. This option is only used by `coqc`.

`-debug`
Switch on the debug flag.

`-emacs`
Tells Coq it is executed under Emacs.

`-db`
Launch Coq under the Objective Caml debugger (provided that Coq has been compiled for debugging; see next chapter).

`-image` ***file***
This option sets the binary image to be used to be *file* instead of the standard one. Not of general use.

`-bindir` ***directory***
Set the directory containing Coq binaries. It is equivalent to do `export COQBIN=`*directory* before lauching Coq.

`-libdir` ***file***
Set the directory containing Coq libraries. It is equivalent to do `export COQLIB=`*directory* before lauching Coq.

`-where`
Print the Coq's standard library location and exit.

`-v`
: Print the Coq's version and exit.

`-h`, `-help`
: Print a short usage and exit.

Chapter 12

Utilities

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

12.1 Building a toplevel extended with user tactics

The native-code version of Coq cannot dynamically load user tactics using Objective Caml code. It is possible to build a toplevel of Coq, with Objective Caml code statically linked, with the tool `coqmktop`.

For example, one can build a native-code Coq toplevel extended with a tactic which source is in `tactic.ml` with the command

```
% coqmktop -opt -o mytop.out tactic.cmx
```

where `tactic.ml` has been compiled with the native-code compiler `ocamlopt`. This command generates an executable called `mytop.out`. To use this executable to compile your Coq files, use `coqc -image mytop.out`.

A basic example is the native-code version of Coq (`coqtop.opt`), which can be generated by `coqmktop -opt -o coqopt.opt`.

Application: how to use the Objective Caml debugger with Coq. One useful application of `coqmktop` is to build a Coq toplevel in order to debug your tactics with the Objective Caml debugger. You need to have configured and compiled Coq for debugging (see the file `INSTALL` included in the distribution). Then, you must compile the Caml modules of your tactic with the option `-g` (with the bytecode compiler) and build a stand-alone bytecode toplevel with the following command:

`% coqmktop -g -o coq-debug` <*your .cmo files*>

To launch the Objective Caml debugger with the image you need to execute it in an environment which correctly sets the `COQLIB` variable. Moreover, you have to indicate the directories in which `ocamldebug` should search for Caml modules.

A possible solution is to use a wrapper around `ocamldebug` which detects the executables containing the word `coq`. In this case, the debugger is called with the required additional arguments. In other cases, the debugger is simply called without additional arguments. Such a wrapper can be found in the `dev/` subdirectory of the sources.

12.2 Modules dependencies

In order to compute modules dependencies (so to use `make`), Coq comes with an appropriate tool, `coqdep`.

`coqdep` computes inter-module dependencies for Coq and Objective Caml programs, and prints the dependencies on the standard output in a format readable by make. When a directory is given as argument, it is recursively looked at.

Dependencies of Coq modules are computed by looking at `Require` commands (`Require`, `Require Export`, `Require Import`, `Require Implementation`), but also at the command `Declare ML Module`.

Dependencies of Objective Caml modules are computed by looking at `open` commands and the dot notation *module.value*. However, this is done approximatively and you are advised to use `ocamldep` instead for the Objective Caml modules dependencies.

See the man page of `coqdep` for more details and options.

12.3 Creating a `Makefile` for Coq modules

When a proof development becomes large and is split into several files, it becomes crucial to use a tool like `make` to compile Coq modules.

The writing of a generic and complete `Makefile` may be a tedious work and that's why Coq provides a tool to automate its creation, `coq_makefile`. Given the files to compile, the command `coq_makefile` prints a `Makefile` on the standard output. So one has just to run the command:

% coq_makefile $file_1.v \dots file_n.v$ > Makefile

The resulted `Makefile` has a target `depend` which computes the dependencies and puts them in a separate file `.depend`, which is included by the `Makefile`. Therefore, you should create such a file before the first invocation of make. You can for instance use the command

```
% touch .depend
```

Then, to initialize or update the modules dependencies, type in:

```
% make depend
```

There is a target `all` to compile all the files $file_1 \dots file_n$, and a generic target to produce a `.vo` file from the corresponding `.v` file (so you can do `make` *file*`.vo` to compile the file *file*`.v`).

`coq_makefile` can also handle the case of ML files and subdirectories. For more options type

```
% coq_makefile -help
```

Warning: To compile a project containing Objective Caml files you must keep the sources of Coq somewhere and have an environment variable named `COQTOP` that points to that directory.

12.4 Coq and LaTeX

12.4.1 Embedded Coq phrases inside LaTeX documents

When writing a documentation about a proof development, one may want to insert Coq phrases inside a LaTeX document, possibly together with the corresponding answers of the system. We provide a mechanical way to process such Coq phrases embedded in LaTeX files: the `coq-tex` filter. This filter extracts Coq phrases embedded in LaTeX files, evaluates them, and insert the outcome of the evaluation after each phrase.

Starting with a file *file*`.tex` containing Coq phrases, the `coq-tex` filter produces a file named *file*`.v.tex` with the Coq outcome.

There are options to produce the Coq parts in smaller font, italic, between horizontal rules, etc. See the man page of `coq-tex` for more details.

Remark. This Reference Manual and the Tutorial have been completely produced with `coq-tex`.

12.4.2 Documenting Coq files with LaTeX

`coqweb` is a "literate programming" tool for Coq, inspired by Knuth's WEB tool. The user documents his or her files with LaTeX material inside Coq comments and `coqweb` produces a LaTeX document from this unique source. Coq parts are displayed in a nice way (`->` becomes $\rightarrow$, keywords are typeset in a bold face, etc.). Additionally, an index is produced which gives the places where the various globals are introduced.

`coqweb` is developped and distributed independently of the system Coq. It is freely available, with sources, binaries and a full documentation, at `www.lri.fr/~filliatr/coqweb`.

12.5 Coq and HTML

An HTML output can be obtained from Coq files documented using `coqweb` (see the previous paragraph). See the documentation of `coqweb` for more details.

12.6 Coq and GNU Emacs

12.6.1 The Coq Emacs mode

Coq comes with a Major mode for GNU Emacs, `coq.el`. This mode provides syntax highlighting (assuming your GNU Emacs library provides `hilit19.el`) and also a rudimentary indentation facility in the style of the Caml GNU Emacs mode.

Add the following lines to your `.emacs` file:

```
(setq auto-mode-alist (cons '("\\.v$" . coq-mode) auto-mode-alist))
(autoload 'coq-mode "coq" "Major mode for editing Coq vernacular." t)
```

The Coq major mode is triggered by visiting a file with extension `.v`, or manually with the command `M-x coq-mode`. It gives you the correct syntax table for the Coq language, and also a rudimentary indentation facility:

- pressing TAB at the beginning of a line indents the line like the line above;

- extra TABs increase the indentation level (by 2 spaces by default);
- M-TAB decreases the indentation level.

An inferior mode to run Coq under Emacs, by Marco Maggesi, is also included in the distribution, in file `coq-inferior.el`. Instructions to use it are contained in this file.

12.6.2 Proof General

Proof General is a generic interface for proof assistants based on Emacs (or XEmacs). The main idea is that the Coq commands you are editing are sent to a Coq toplevel running behind Emacs and the answers of the system automatically inserted into other Emacs buffers. Thus you don't need to copy-paste the Coq material from your files to the Coq toplevel or conversely from the Coq toplevel to some files.

Proof General is developped and distributed independently of the system Coq. It is freely available at `www.proofgeneral.org`.

12.7 Module specification

Given a Coq vernacular file, the `gallina` filter extracts its specification (inductive types declarations, definitions, type of lemmas and theorems), removing the proofs parts of the file. The Coq file *file*`.v` gives birth to the specification file *file*`.g` (where the suffix `.g` stands for Gallina).

See the man page of `gallina` for more details and options.

12.8 Man pages

There are man pages for the commands `coqdep`, `gallina` and `coq-tex`. Man pages are installed at installation time (see installation instructions in file `INSTALL`, step 6).

The Coq Proof Assistant

Addenddum to the Reference Manual

September 16, 2002

Version 7.3 [1]

LogiCal Project

[1]This research was partly supported by ESPRIT Basic Research Action "Types"

V7.3, September 16, 2002

Presentation of the Addendum

Here you will find several pieces of additional documentation for the Coq Reference Manual. Each of this chapters is concentrated on a particular topic, that should interest only a fraction of the Coq users : that's the reason why they are apart from the Reference Manual.

Extended pattern-matching This chapter details the use of generalized pattern-matching. It is contributed by Cristina Cornes and Hugo Herbelin

Implicit coercions This chapter details the use of the coercion mechanism. It is contributed by Amokrane Saïbi.

Proof of imperative programs This chapter explains how to prove properties of annotated programs with imperative features. It is contributed by Jean-Christophe Filliâtre

Program extraction This chapter explains how to extract in practice ML files from F_ω terms. It is contributed by Jean-Christophe Filliâtre and Pierre Letouzey.

Omega `Omega`, written by Pierre Crégut, solves a whole class of arithmetic problems.

Program The `Program` technology intends to inverse the extraction mechanism. It allows the developments of certified programs in Coq. This chapter is due to Catherine Parent. **This feature is not available in Coq version 7.**

The `Ring` tactic This is a tactic to do AC rewriting. This chapter explains how to use it and how it works. The chapter is contributed by Patrick Loiseleur.

The `Setoid_replace` tactic This is a tactic to do rewriting on types equipped with specific (only partially substitutive) equality. The chapter is contributed by Clément Renard.

Contents

Chapter 13

Extended pattern-matching

Cristina Cornes

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

13.1 Patterns

The full syntax of `Cases` is presented in figure 13.1. Identifiers in patterns are either constructor names or variables. Any identifier that is not the constructor of an inductive or coinductive type is considered to be a variable. A variable name cannot occur more than once in a given pattern. It is recommended to start variable names by a lowercase letter.

If a pattern has the form $(c\ \vec{x})$ where c is a constructor symbol and $\vec{x}$ is a linear vector of variables, it is called *simple*: it is the kind of pattern recognized by the basic version of `Cases`. If a pattern is not simple we call it *nested*.

A variable pattern matches any value, and the identifier is bound to that value. The pattern "_" (called "don't care" or "wildcard" symbol) also matches any value, but does not bind anything. It may occur an arbitrary number of times in a pattern. Alias patterns written `(`*pattern* `as` *identifier*`)` are also accepted. This pattern matches the same values as *pattern* does and *identifier* is bound to the matched value. A list of patterns is also considered as a pattern and is called *multiple pattern*.

Notice also the annotation is mandatory when the sequence of equation is empty.

Since extended `Cases` expressions are compiled into the primitive ones, the expressiveness of the theory remains the same. Once the stage of parsing has finished only simple patterns remain. An easy way to see the result of the expansion is by printing the term with `Print` if the term is a constant, or using the command `Check`.

The extended `Cases` still accepts an optional *elimination predicate* enclosed between brackets `<>`. Given a pattern matching expression, if all the right hand sides of `=>` (*rhs* in short) have the same type, then this type can be sometimes synthesized, and so we can omit the `<>`. Otherwise the predicate between `<>` has to be provided, like for the basic `Cases`.

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:

nested_pattern	:=	*ident*
	|	_
	|	(*ident nested_pattern* ... *nested_pattern*)
	|	(*nested_pattern* `as` *ident*)
	|	(*nested_pattern* , *nested_pattern*)
	|	(*nested_pattern*)
mult_pattern	:=	nested_pattern ... nested_pattern
ext_eqn	:=	*mult_pattern* => *term*
term	:=	*[annotation]* `Cases` *term* ... *term* `of` *[ext_eqn* | ... | *ext_eqn]* `end`

Figure 13.1: Extended Cases syntax

```
Coq < Fixpoint max [n,m:nat] : nat :=
Coq <    Cases n of
Coq <       O     => m
Coq <    | (S n') => Cases m of
Coq <                   O     => (S n')
Coq <                 | (S m') => (S (max n' m'))
Coq <                 end
Coq <    end.
max is recursively defined
```

Using multiple patterns in the definition allows to write:

```
Coq < Reset max.

Coq < Fixpoint max [n,m:nat] : nat :=
Coq <    Cases n m of
Coq <       O          _  => m
Coq <    | (S n') O       => (S n')
Coq <    | (S n') (S m') => (S (max n' m'))
Coq <    end.
max is recursively defined
```

which will be compiled into the previous form.

The pattern-matching compilation strategy examines patterns from left to right. A `Cases` 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 `Cases` expression.

```
Coq < Check [x:nat]<nat>Cases x of y => y end.
[x:nat]x
     : nat->nat
```

We can also use "`as` patterns" to associate a name to a sub-pattern:

```
Coq < Reset max.

Coq < Fixpoint max [n:nat] : nat -> nat :=
```

```
Coq <   [m:nat] Cases n m of
Coq <              O      _          => m
Coq <            | ((S n') as p) O  => p
Coq <            | (S n') (S m')    => (S (max n' m'))
Coq <           end.
max is recursively defined
```

Here is now an example of nested patterns:

```
Coq < Fixpoint even [n:nat] : bool :=
Coq <    Cases n of
Coq <       O          => true
Coq <    | (S O)       => false
Coq <    | (S (S n')) => (even n')
Coq <    end.
even is recursively defined
```

This is compiled into:

```
Coq < Print even.
even =
Fix even
  {even [n:nat] : bool :=
     Cases n of
       O => true
     | (S n0) => Cases n0 of
                   O => false
                 | (S n') => (even n')
                 end
     end}
     : nat->bool
```

In the previous examples patterns do not conflict with, but sometimes it is comfortable to write patterns that admit a non trivial superposition. Consider the boolean function `lef` that given two natural numbers yields `true` if the first one is less or equal than the second one and `false` otherwise. We can write it as follows:

```
Coq < Fixpoint lef [n,m:nat] : bool :=
Coq <           Cases n m of
Coq <               O     x     => true
Coq <            |  x     O     => false
Coq <            | (S n) (S m)  => (lef n m)
Coq <           end.
lef is recursively defined
```

Note that the first and the second multiple pattern superpose because the couple of values `O O` matches both. Thus, what is the result of the function on those values? To eliminate ambiguity we use the *textual priority rule*: we consider patterns ordered from top to bottom, then a value is matched by the pattern at the ith row if and only if it is not matched by some pattern of a previous row. Thus in the example, `O O` is matched by the first pattern, and so `(lef O O)` yields `true`.

Another way to write this function is:

```
Coq < Reset lef.

Coq < Fixpoint lef [n,m:nat] : bool :=
Coq <          Cases n m of
Coq <              O     x    => true
Coq <           | (S n) (S m) => (lef n m)
Coq <           | _     _     => false
Coq <          end.
lef is recursively defined
```

Here the last pattern superposes with the first two. Because of the priority rule, the last pattern will be used only for values that do not match neither the first nor the second one.

Terms with useless patterns are not accepted by the system. Here is an example:

```
Coq < Check [x:nat]Cases x of O => true | (S _) => false | x => true end.
Toplevel input, characters 53-62
> Check [x:nat]Cases x of O => true | (S _) => false | x => true end.
>                                                       ^^^^^^^^^
Error: This clause is redundant
```

13.2 About patterns of parametric types

When matching objects of a parametric type, constructors in patterns *do not expect* the parameter arguments. Their value is deduced during expansion.

Consider for example the polymorphic lists:

```
Coq < Inductive List [A:Set] :Set :=
Coq <   nil:(List A)
Coq < | cons:A->(List A)->(List A).
List is defined
List_rect is defined
List_ind is defined
List_rec is defined
```

We can check the function *tail*:

```
Coq < Check [l:(List nat)]Cases l of
Coq <                                nil         => (nil nat)
Coq <                              | (cons _ l') => l'
Coq <                              end.
[l:(List nat)]Cases l of
                nil => (nil nat)
              | (cons _ l') => l'
              end
     : (List nat)->(List nat)
```

When we use parameters in patterns there is an error message:

```
Coq < Check [l:(List nat)]Cases l of
Coq <                                (nil A)       => (nil nat)
Coq <                              | (cons A _ l') => l'
Coq <                              end.
Toplevel input, characters 116-120
>                             | (cons A _ l') => l'
>                                ^^^^
Error: The constructor cons expects 2 arguments.
```

13.3 Matching objects of dependent types

The previous examples illustrate pattern matching on objects of non-dependent types, but we can also use the expansion strategy to destructure objects of dependent type. Consider the type `listn` of lists of a certain length:

```
Coq < Inductive listn : nat-> Set :=
Coq <   niln : (listn O)
Coq < | consn : (n:nat)nat->(listn n) -> (listn (S n)).
listn is defined
listn_rect is defined
listn_ind is defined
listn_rec is defined
```

13.3.1 Understanding dependencies in patterns

We can define the function `length` over `listn` by:

```
Coq < Definition length := [n:nat][l:(listn n)] n.
length is defined
```

Just for illustrating pattern matching, we can define it by case analysis:

```
Coq < Reset length.

Coq < Definition length := [n:nat][l:(listn n)]
Coq <                        Cases l of
Coq <                           niln         => O
Coq <                        | (consn n _ _) => (S n)
Coq <                        end.
length is defined
```

We can understand the meaning of this definition using the same notions of usual pattern matching.

13.3.2 When the elimination predicate must be provided

The examples given so far do not need an explicit elimination predicate between <> because all the rhs have the same type and the strategy succeeds to synthesize it. Unfortunately when dealing with dependent patterns it often happens that we need to write cases where the type of the rhs are different instances of the elimination predicate. The function `concat` for `listn` is an example where the branches have different type and we need to provide the elimination predicate:

```
Coq < Fixpoint concat [n:nat; l:(listn n)]
Coq <  :  (m:nat) (listn m) -> (listn (plus n m))
Coq <  := [m:nat][l':(listn m)]
Coq <       <[n:nat](listn (plus n m))>Cases l  of
Coq <          niln          => l'
Coq <       | (consn n' a y) => (consn (plus n' m) a (concat n' y m l'))
Coq <       end.
concat is recursively defined
```

Recall that a list of patterns is also a pattern. So, when we destructure several terms at the same time and the branches have different type we need to provide the elimination predicate for this multiple pattern.

For example, an equivalent definition for `concat` (even though the matching on the second term is trivial) would have been:

```
Coq < Reset concat.

Coq < Fixpoint concat [n:nat; l:(listn n)]
Coq <      : (m:nat) (listn m) -> (listn (plus n m)) :=
Coq <   [m:nat][l':(listn m)]
Coq <     <[n,_:nat](listn (plus n m))>Cases l l' of
Coq <           niln          x => x
Coq <         | (consn n' a y) x => (consn (plus n' m) a (concat n' y m x))
Coq <                       end.
concat is recursively defined
```

Notice that this time, the predicate `[n,_:nat](listn (plus n m))` is binary because we destructure both `l` and `l'` whose types have arity one. In general, if we destructure the terms $e_1 \ldots e_n$ the predicate will be of arity m where m is the sum of the number of dependencies of the type of $e_1, e_2, \ldots e_n$ (the λ-abstractions should correspond from left to right to each dependent argument of the type of $e_1 \ldots e_n$). When the arity of the predicate (i.e. number of abstractions) is not correct Coq raises an error message. For example:

```
Coq < Fixpoint concat [n:nat; l:(listn n)]
Coq <      : (m:nat) (listn m) -> (listn (plus n m)) :=
Coq <   [m:nat][l':(listn m)]
Coq <    <[n:nat](listn (plus n m))>Cases l l' of
Coq <          | niln          x => x
Coq <          | (consn n' a y) x => (consn (plus n' m) a (concat n' y m x))
Coq <          end.
Toplevel input, characters 119-120
>    <[n:nat](listn (plus n m))>Cases l l' of
>      ^
Error: The elimination predicate
[n:nat](listn (plus n m))
should be of arity nat->nat->Type (for non dependent case) or
(n:nat)(listn n)->(n0:nat)(listn n0)->Type (for dependent case).
```

13.4 Using pattern matching to write proofs

In all the previous examples the elimination predicate does not depend on the object(s) matched. But it may depend and the typical case is when we write a proof by induction or a function that yields an object of dependent type. An example of proof using `Cases` in given in section 8.1

For example, we can write the function `buildlist` that given a natural number n builds a list of length n containing zeros as follows:

```
Coq < Fixpoint buildlist [n:nat] : (listn n) :=
Coq <     <[n:nat](listn n)>Cases n of
Coq <             O    => niln
Coq <         |  (S n) => (consn n O (buildlist n))
```

```
Coq <        end.
buildlist is recursively defined
```

We can also use multiple patterns whenever the elimination predicate has the correct arity.

Consider the following definition of the predicate less-equal `Le`:

```
Coq < Inductive LE : nat->nat->Prop :=
Coq <   LEO: (n:nat)(LE O n)
Coq < | LES: (n,m:nat)(LE n m) -> (LE (S n) (S m)).
LE is defined
LE_ind is defined
```

We can use multiple patterns to write the proof of the lemma `(n,m:nat) (LE n m)\/(LE m n)`:

```
Coq < Fixpoint dec  [n:nat] : (m:nat)(LE n m) \/ (LE m n) :=
Coq <  [m:nat] <[n,m:nat](LE n m) \/ (LE m n)>Cases n m of
Coq <            O   x =>  (or_introl ? (LE x O) (LEO x))
Coq <          | x   O =>  (or_intror (LE x O) ? (LEO x))
Coq <          | ((S n) as n') ((S m) as m') =>
Coq <               Cases (dec n m) of
Coq <                   (or_introl h) => (or_introl ? (LE m' n') (LES n m h))
Coq <                 | (or_intror h) => (or_intror (LE n' m') ? (LES m n h))
Coq <               end
Coq <         end.
dec is recursively defined
```

In the example of `dec` the elimination predicate is binary because we destructure two arguments of `nat` which is a non-dependent type. Notice that the first `Cases` is dependent while the second is not.

In general, consider the terms $e_1 \dots e_n$, where the type of e_i is an instance of a family type $[\vec{d_i} : \vec{D_i}]T_i$ $(1 \leq i \leq n)$. Then, in expression `<`$\mathcal{P}$`>Cases` $e_1 \dots e_n$ `of` $\dots$ `end`, the elimination predicate $\mathcal{P}$ should be of the form: $[\vec{d_1} : \vec{D_1}][x_1 : T_1] \dots [\vec{d_n} : \vec{D_n}][x_n : T_n]Q$.

The user can also use `Cases` in combination with the tactic `Refine` (see section 7.2.2) to build incomplete proofs beginning with a `Cases` construction.

13.5 Pattern-matching on inductive objects involving local definitions

If local definitions occur in the type of a constructor, then there are two ways to match on this constructor. Either the local definitions are skipped and matching is done only on the true arguments of the constructors, or the bindings for local definitions can also be caught in the matching.

Example.

```
Coq < Inductive list : nat -> Set :=
Coq < | nil : (list O)
Coq < | cons : (n:nat)[m:=(mult (2) n)](list m)->(list (S (S m))).
```

In the next example, the local definition is not caught.

```
Coq < Fixpoint length [n; l:(list n)] : nat :=
Coq <   Cases l of
Coq <      nil => O
Coq <   | (cons n l0) => (S (length (mult (2) n) l0))
Coq <   end.
length is recursively defined
```

But in this example, it is.

```
Coq < Fixpoint length' [n; l:(list n)] : nat :=
Coq <   Cases l of
Coq <      nil => O
Coq <   | (cons _ m l0) => (S (length' m l0))
Coq <   end.
length' is recursively defined
```

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

13.6 Pattern-matching and coercions

If a mismatch occurs between the expected type of a pattern and its actual type, a coercion made from constructors is sought. If such a coercion can be found, it is automatically inserted around the pattern.

Example:

```
Coq < Inductive I : Set :=
Coq <   C1 : nat -> I
Coq < | C2 : I -> I.
I is defined
I_rect is defined
I_ind is defined
I_rec is defined

Coq < Coercion C1 : nat >-> I.
C1 is now a coercion

Coq < Check [x]Cases x of (C2 O) => O | _ => O end.
[x:I]
 Cases x of
   (C1 _) => (0)
 | (C2 i) =>
    Cases i of
      (C1 n) => Cases n of
                  0 => (0)
                | (S _) => (0)
                end
    | (C2 _) => (0)
    end
 end
     : I->nat
```

13.7 When does the expansion strategy fail ?

The strategy works very like in ML languages when treating patterns of non-dependent type. But there are new cases of failure that are due to the presence of dependencies.

The error messages of the current implementation may be sometimes confusing. When the tactic fails because patterns are somehow incorrect then error messages refer to the initial expression. But the strategy may succeed to build an expression whose sub-expressions are well typed when the whole expression is not. In this situation the message makes reference to the expanded expression. We encourage users, when they have patterns with the same outer constructor in different equations, to name the variable patterns in the same positions with the same name. E.g. to write `(cons n O x) => e1` and `(cons n _ x) => e2` instead of `(cons n O x) => e1` and `(cons n' _ x') => e2`. This helps to maintain certain name correspondence between the generated expression and the original.

Here is a summary of the error messages corresponding to each situation:

- patterns are incorrect (because constructors are not applied to the correct number of the arguments, because they are not linear or they are wrongly typed)
  - `The constructor` *ident* `expects` *num* `arguments`
  - `The variable` *ident* `is bound several times in pattern` *term*
  - `Found a constructor of inductive type` *term* `while a constructor of` *term* `is expected`
- the pattern matching is not exhaustive
  - `Non exhaustive pattern-matching`
- the elimination predicate provided to `Cases` has not the expected arity
  - `The elimination predicate` *term* `should be of arity` *num* `(for non dependent case) or` *num* `(for dependent case)`
- the whole expression is wrongly typed
- there is a type mismatch between the different branches
  - `Unable to infer a Cases predicate`
`Either there is a type incompatiblity or the problem involves dependencies`

  Then the user should provide an elimination predicate.

Chapter 14

Implicit Coercions

Amokrane Saïbi

14.1 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 $f : (x : A)B$ 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.

14.2 Classes

A class with n parameters is any defined name with a type $(x_1 : A_1)..(x_n : A_n)s$ where s is a sort. Thus a class with parameters is considered as a single class and not as a family of classes. An object of a class C is any term of type $(C\ t_1..t_n)$. In addition to these user-classes, we have two abstract classes:

- `SORTCLASS`, the class of sorts; its objects are the terms whose type is a sort.
- `FUNCLASS`, the class of functions; its objects are all the terms with a functional type, i.e. of form $(x : A)B$.

Formally, the syntax of a classes is defined by

class	::=	*qualid*
	|	`SORTCLASS`
	|	`FUNCLASS`

14.3 Coercions

A name f can be declared as a coercion between a source user-class C with n parameters and a target class D if one of these conditions holds:

- D is a user-class, then the type of f must have the form $(x_1 : A_1)..(x_n : A_n)(y : (C\ x_1..x_n))(D\ u_1..u_m)$ where m is the number of parameters of D.
- D is `FUNCLASS`, then the type of f must have the form $(x_1 : A_1)..(x_n : A_n)(y : (C\ x_1..x_n))(x : A)B$.
- D is `SORTCLASS`, then the type of f must have the form $(x_1 : A_1)..(x_n : A_n)(y : (C\ x_1..x_n))s$ with s a sort.

We then write $f : C{>}{-}{>}D$. The restriction on the type of coercions is called *the uniform inheritance condition*. Remark that the abstract classes `FUNCLASS` and `SORTCLASS` cannot be source classes.

To coerce an object $t : (C\ t_1..t_n)$ of C towards D, we have to apply the coercion f to it; the obtained term $(f\ t_1..t_n\ t)$ is then an object of D.

14.4 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 $f : (x_1 : T_1)..(x_k : T_k)(y : (C\ u_1..u_n))(D\ v_1..v_m)$ a function which does not verify the uniform inheritance condition. To declare f as coercion, one has first to declare a subclass C' of C:

$$C' := [x_1 : T_1]..[x_k : T_k](C\ u_1..u_n)$$

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

$$\begin{array}{lcl} Id\_C'\_C & := & [x_1 : T_1]..[x_k : T_k][y : (C'\ x_1..x_k)] \\ & & (y :: (C\ u_1..u_n)) \end{array}$$

We can now declare f as coercion from C' to D, since we can "cast" its type as $(x_1 : T_1)..(x_k : T_k)(y : (C'\ x_1..x_k))(D\ v_1..v_m)$.
The identity coercions have a special status: to coerce an object $t : (C'\ t_1..t_k)$ of C' towards C, we have not to insert explicitly $Id\_C'\_C$ since $(Id\_C'\_C\ t_1..t_k\ t)$ is convertible with t. However we "rewrite" the type of t to become an object of C; in this case, it becomes $(C\ u_1^*..u_k^*)$ where each u_i^* is the result of the substitution in u_i of the variables x_j by t_j.

14.5 Inheritance Graph

Coercions form an inheritance graph with classes as nodes. We call *coercion path* an ordered list of coercions between two nodes of the graph. A class C is said to be a subclass of D if there is a coercion path in the graph from C to D; we also say that C inherits from D. Our mechanism supports multiple inheritance since a class may inherit from several classes, contrary to simple inheritance where a class inherits from at most one class. However there must be at most one path between two classes. If this is not the case, only the *oldest* one is valid and the others are ignored. So the order of declaration of coercions is important.

We extend notations for coercions to coercion paths. For instance $[f_1; ..; f_k]$: C`>->`D is the coercion path composed by the coercions $f_1..f_k$. The application of a coercion path to a term consists of the successive application of its coercions.

14.6 Declaration of Coercions

14.6.1 `Coercion` *qualid* `:` *class*$_1$ `>->` *class*$_2$`.`

Declares the construction denoted by *qualid* as a coercion between *class*$_1$ and *class*$_2$.

Error messages:

1. *qualid* `not declared`
2. *qualid* `is already a coercion`
3. `FUNCLASS cannot be a source class`
4. `SORTCLASS cannot be a source class`
5. `Does not correspond to a coercion`
   qualid is not a function.
6. `Cannot find the source class`
7. *qualid* `does not respect the inheritance uniform condition`
8. `The target class does not correspond to` *class*$_2$

When the coercion *qualid* is added to the inheritance graph, non valid coercion paths are ignored; they are signaled by a warning.

Warning :

1. `Ambiguous paths:` $[f_1^1; ..; f_{n_1}^1] : C_1$`>->`$D_1$
   ...
   $[f_1^m; ..; f_{n_m}^m] : C_m$`>->`$D_m$

Variants:

1. `Coercion Local` *qualid* `:` *class*$_1$ `>->` *class*$_2$`.`
   Declares the construction denoted by *qualid* as a coercion local to the current section.

2. `Coercion` *ident* `:=` *term*
   This defines *ident* just like `Definition` *ident* `:=` *term*, and then declares *ident* as a coercion between it source and its target.

3. `Coercion` *ident* `:=` *term* `:` *type*
   This defines *ident* just like `Definition` *ident* `:` *type* `:=` *term*, and then declares *ident* as a coercion between it source and its target.

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

14.6.2 `Identity Coercion` *ident*`:`$class_1$ `>->` $class_2$`.`

We check that $class_1$ is a constant with a value of the form $[x_1 : T_1]..[x_n : T_n](class_2\ t_1..t_m)$ where m is the number of parameters of $class_2$. Then we define an identity function with the type $(x_1 : T_1)..(x_n : T_n)(y : (class_1\ x_1..x_n))(class_2\ t_1..t_m)$, and we declare it as an identity coercion between $class_1$ and $class_2$.

Error messages:

1. `Clash with previous constant` *ident*

2. $class_1$ `must be a transparent constant`

Variants:

1. `Identity Coercion Local` *ident*`:`$ident_1$ `>->` $ident_2$`.`
   Idem but locally to the current section.

14.7 Displaying Available Coercions

14.7.1 `Print Classes.`

Print the list of declared classes in the current context.

14.7.2 `Print Coercions.`

Print the list of declared coercions in the current context.

14.7.3 `Print Graph.`

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

14.7.4 `Print Coercion Paths` $class_1$ $class_2$`.`

Print the list of valid coercion paths from $class_1$ to $class_2$.

14.8 Activating the Printing of Coercions

14.8.1 `Set Printing Coercions.`

This command forces all the coercions to be printed. To skip the printing of coercions, use `Unset Printing Coercions`. By default, coercions are not printed.

14.8.2 `Set Printing Coercion` *qualid* `.`

This command forces coercion denoted by *qualid* to be printed. To skip the printing of coercion *qualid*, use `Unset Printing Coercion` *qualid*. By default, a coercion is never printed.

14.9 Classes as Records

We allow the definition of *Structures with Inheritance* (or classes as records) by extending the existing `Record` macro (see section 2.1). Its new syntax is:

`Record` *[>] ident* `[` *params* `]` `:` *sort* `:=` *[ident$_0$]* `{`
    ident$_1$ [`:`|`:>`] *term*$_1$ `;`
    ...
    ident$_n$ [`:`|`:>`] *term*$_n$ `}.`

The identifier *ident* is the name of the defined record and *sort* is its type. The identifier *ident*$_0$ is the name of its constructor. The identifiers *ident*$_1$, .., *ident*$_n$ are the names of its fields and *term*$_1$, .., *term*$_n$ their respective types. The alternative [`:`|`:>`] is "`:`" or "`:>`". If *ident*$_i$`:>`*term*$_i$, then *ident*$_i$ is automatically declared as coercion from *ident* to the class of *term*$_i$. Remark that *ident*$_i$ always verifies the uniform inheritance condition. If the optional ">" before *ident* is present, then *ident*$_0$ (or the default name `Build_`*ident* if *ident*$_0$ is omitted) is automatically declared as a coercion from the class of *term*$_n$ to *ident* (this may fail if the uniform inheritance condition is not satisfied).

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

14.10 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 (no warning message is printed). Coercions with a local source class or a local target class, and coercions which do no more verify the uniform inheritance condition are also forgotten.

14.11 Examples

There are three situations:

- $(f\ a)$ is ill-typed where $f : (x : A)B$ and $a : A'$. If there is a coercion path between A' and A, $(f\ a)$ is transformed into $(f\ a')$ where a' is the result of the application of this coercion path to a.

  We first give an example of coercion between atomic inductive types

```
Coq < Definition bool_in_nat := [b:bool]if b then O else (S O).
bool_in_nat is defined

Coq < Coercion bool_in_nat : bool >-> nat.
bool_in_nat is now a coercion

Coq < Check O=true.
O=true
     : Prop

Coq < Set Printing Coercions.

Coq < Check O=true.
O=(bool_in_nat true)
     : Prop
```

Warning: "`Check true=O.`" fails. This is "normal" behaviour of coercions. To validate `true=O`, the coercion is searched from `nat` to `bool`. There is no one.

We give an example of coercion between classes with parameters.

```
Coq < Parameters C:nat->Set; D:nat->bool->Set; E:bool->Set.
C is assumed
D is assumed
E is assumed

Coq < Parameter f : (n:nat)(C n) -> (D (S n) true).
f is assumed

Coq < Coercion f : C >-> D.
f is now a coercion

Coq < Parameter g : (n:nat)(b:bool)(D n b) -> (E b).
g is assumed

Coq < Coercion g : D >-> E.
g is now a coercion

Coq < Parameter c : (C O).
c is assumed

Coq < Parameter T : (E true) -> nat.
T is assumed

Coq < Check (T c).
(T c)
     : nat

Coq < Set Printing Coercions.

Coq < Check (T c).
(T (g (S O) true (f O c)))
     : nat
```

We give now an example using identity coercions.

```
Coq < Definition D' := [b:bool](D (S O) b).
D' is defined
```

```
Coq < Identity Coercion IdD'D : D' >-> D.

Coq < Print IdD'D.
IdD'D =
([b:bool; x:(D' b)]x)::((b:bool)(D' b)->(D (S O) b))
     : (b:bool)(D' b)->(D (S O) b)

Coq < Parameter d' : (D' true).
d' is assumed

Coq < Check (T d').
(T d')
     : nat

Coq < Set Printing Coercions.

Coq < Check (T d').
(T (g (S O) true d'))
     : nat
```

In the case of functional arguments, we use the monotonic rule of sub-typing. Approximatively, to coerce $t : (x : A)B$ towards $(x : A')B'$, one have to coerce A' towards A and B towards B'. An example is given below:

```
Coq < Parameters A,B:Set; h:A->B.
A is assumed
B is assumed
h is assumed

Coq < Coercion h : A >-> B.
h is now a coercion

Coq < Parameter U : (A -> (E true)) -> nat.
U is assumed

Coq < Parameter t : B -> (C O).
t is assumed

Coq < Check (U t).
(U [x:A](t x))
     : nat

Coq < Set Printing Coercions.

Coq < Check (U t).
(U [x:A](g (S O) true (f O (t (h x)))))
     : nat
```

Remark the changes in the result following the modification of the previous example.

```
Coq < Parameter U' : ((C O) -> B) -> nat.
U' is assumed

Coq < Parameter t' : (E true) -> A.
t' is assumed

Coq < Check (U' t').
(U' [x:(C O)](t' x))
```

```
     : nat
Coq < Set Printing Coercions.
Coq < Check (U' t').
(U' [x:(C O)](h (t' (g (S O) true (f O x)))))
     : nat
```

- An assumption $x : A$ when A is not a type, is ill-typed. It is replaced by $x : A'$ where A' is the result of the application to A of the coercion path between the class of A and SORTCLASS if it exists. This case occurs in the abstraction $[x : A]t$, universal quantification $(x : A)B$, global variables and parameters of (co-)inductive definitions and functions. In $(x : A)B$, such a coercion path may be applied to B also if necessary.

```
Coq < Parameter Graph : Type.
Graph is assumed
Coq < Parameter Node : Graph -> Type.
Node is assumed
Coq < Coercion Node : Graph >-> SORTCLASS.
Node is now a coercion
Coq < Parameter G : Graph.
G is assumed
Coq < Parameter Arrows : G -> G -> Type.
Arrows is assumed
Coq < Check Arrows.
Arrows
     : G->G->Type
Coq < Parameter fg : G -> G.
fg is assumed
Coq < Check fg.
fg
     : G->G
Coq < Set Printing Coercions.
Coq < Check fg.
fg
     : (Node G)->(Node G)
```

- $(f\ a)$ is ill-typed because $f : A$ is not a function. The term f is replaced by the term obtained by applying to f the coercion path between A and FUNCLASS if it exists.

```
Coq < Parameter bij : Set -> Set -> Set.
bij is assumed
Coq < Parameter ap : (A,B:Set)(bij A B) -> A -> B.
ap is assumed
Coq < Coercion ap : bij >-> FUNCLASS.
ap is now a coercion
Coq < Parameter b : (bij nat nat).
```

```
b is assumed

Coq < Check (b O).
(b O)
     : nat

Coq < Set Printing Coercions.

Coq < Check (b O).
(ap nat nat b O)
     : nat
```

Let us see the resulting graph of this session.

```
Coq < Print Graph.
[ap] : bij >-> FUNCLASS
[Node] : Graph >-> SORTCLASS
[h] : A >-> B
[IdD'D; g] : D' >-> E
[IdD'D] : D' >-> D
[f; g] : C >-> E
[g] : D >-> E
[f] : C >-> D
[bool_in_nat] : bool >-> nat
```

Chapter 15

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

Pierre Crégut

15.1 Description of Omega

Omega solves a goal in Presburger arithmetic, ie a universally quantified formula made of equations and inequations. Equations may be specified either on the type nat of natural numbers or on the type Z of binary-encoded integer numbers. Formulas on nat are automatically injected into Z. The procedure may use any hypothesis of the current proof session to solve the goal.

Multiplication is handled by Omega but only goals where at least one of the two multiplicands of products is a constant are solvable. This is the restriction meaned by "Presburger arithmetic".

If the tactic cannot solve the goal, it fails with an error message. In any case, the computation eventually stops.

15.1.1 Arithmetical goals recognized by Omega

Omega applied only to quantifier-free formulas built from the connectors

```
/\, \/, ~, ->
```

on atomic formulas. Atomic formulas are built from the predicates

```
=, le, lt, gt, ge
```

on nat or from the predicates

```
=, <, <=, >, >=
```

on Z. In expressions of type nat, Omega recognizes

```
plus, minus, mult, pred, S, O
```

and in expressions of type `Z`, `Omega` recognizes

`+, -, *, Zs`, and constants.

All expressions of type `nat` or `Z` not built on these operators are considered abstractly as if they were arbitrary variables of type `nat` or `Z`.

15.1.2 Messages from `Omega`

When `Omega` does not solve the goal, one of the following errors is generated:

Error messages:

1. `Omega can't solve this system`
   This may happen if your goal is not quantifier-free (if it is universally quantified, try `Intros` first; if it contains existentials quantifiers too, `Omega` is not strong enough to solve your goal). This may happen also if your goal contains arithmetical operators unknown from `Omega`. Finally, your goal may be really wrong !
2. `Omega: Not a quantifier-free goal`
   If your goal is universally quantified, you should first apply `Intro` as many time as needed.
3. `Omega: Unrecognized predicate or connective:`*ident*
4. `Omega: Unrecognized atomic proposition:`*prop*
5. `Omega: Can't solve a goal with proposition variables`
6. `Omega: Unrecognized proposition`
7. `Omega: Can't solve a goal with non-linear products`
8. `Omega: Can't solve a goal with equality on` *type*

Use `Set Omega` *flag* to set the flag *flag*. Use `Unset Omega` *flag* to unset it and `Switch Omega` *flag* to toggle it.

15.2 Using `Omega`

The tactic `Omega` does not belong to the core system. It should be loaded by

```
Coq < Require Omega.
```

Example 6:

```
Coq < Goal (m,n:Z) ~ `1+2*m = 2*n`.
1 subgoal

  ============================
   (m,n:Z)`1+2*m <> 2*n`

Coq < Intros; Omega.
Subtree proved!
```

Example 7:

```
Coq < Goal (z:Z)`z>0` -> `2*z + 1 > z`.
1 subgoal

  ============================
   (z:Z)`z > 0`->`2*z+1 > z`

Coq < Intro; Omega.
Subtree proved!
```

15.3 Technical data

15.3.1 Overview of the tactic

- The goal is negated twice and the first negation is introduced as an hypothesis.
- Hypothesis are decomposed in simple equations or inequations. Multiple goals may result from this phase.
- Equations and inequations over `nat` are translated over `Z`, multiple goals may result from the translation of substraction.
- Equations and inequations are normalized.
- Goals are solved by the *OMEGA* decision procedure.
- The script of the solution is replayed.

15.3.2 Overview of the *OMEGA* decision procedure

The *OMEGA* decision procedure involved in the `Omega` tactic uses a small subset of the decision procedure presented in

> "The Omega Test: a fast and practical integer programming algorithm for dependence analysis", William Pugh, Communication of the ACM , 1992, p 102-114.

Here is an overview. The reader is refered to the original paper for more information.

- Equations and inequations are normalized by division by the GCD of their coefficients.
- Equations are eliminated, using the Banerjee test to get a coefficient equal to one.
- Note that each inequation defines a half space in the space of real value of the variables.
- Inequations are solved by projecting on the hyperspace defined by cancelling one of the variable. They are partitioned according to the sign of the coefficient of the eliminated variable. Pairs of inequations from different classes define a new edge in the projection.
- Redundant inequations are eliminated or merged in new equations that can be eliminated by the Banerjee test.

- The last two steps are iterated until a contradiction is reached (success) or there is no more variable to eliminate (failure).

It may happen that there is a real solution and no integer one. The last steps of the Omega procedure (dark shadow) are not implemented, so the decision procedure is only partial.

15.4 Bugs

- The simplification procedure is very dumb and this results in many redundant cases to explore.
- Much too slow.
- Certainely other bugs! You can report them to

  `Pierre.Cregut@cnet.francetelecom.fr`

Chapter 16

Proof of imperative programs

Jean-Christophe Filliâtre

This chapter describes a new tactic to prove the correctness and termination of imperative programs annotated in a Floyd-Hoare logic style. The theoretical fundations of this tactic are described in [47, 49]. This tactic is provided in the Coq module `Correctness`, which does not belong to the initial state of Coq. So you must import it when necessary, with the following command:

```
Require Correctness.
```

16.1 How it works

Before going on into details and syntax, let us give a quick overview of how that tactic works. Its behavior is the following: you give a program annotated with logical assertions and the tactic will generate a bundle of subgoals, called *proof obligations*. Then, if you prove all those proof obligations, you will establish the correctness and the termination of the program. The implementation currently supports traditional imperative programs with references and arrays on arbitrary purely functional datatypes, local variables, functions with call-by-value and call-by-variable arguments, and recursive functions.

Although it behaves as an implementation of Floyd-Hoare logic, it is not. The idea of the underlying mechanism is to translate the imperative program into a partial proof of a proposition of the kind

$$\forall \vec{x}.P(\vec{x}) \Rightarrow \exists(\vec{y},v).Q(\vec{x},\vec{y},v)$$

where P and Q stand for the pre- and post-conditions of the program, $\vec{x}$ and $\vec{y}$ the variables used and modified by the program and v its result. Then this partial proof is given to the tactic `Refine` (see 7.2.2, page 112), which effect is to generate as many subgoals as holes in the partial proof term.

The syntax to invoke the tactic is the following:

`Correctness` $ident\ \ annotated\_program.$

Notice that this is not exactly a *tactic*, since it does not apply to a goal. To be more rigorous, it is the combination of a vernacular command (which creates the goal from the annotated program) and a tactic (which partially solves it, leaving some proof obligations to the user).

Although `Correctness` is not a tactic, the following syntax is available:

`Correctness` *ident annotated_program* ; *tactic*.

In that case, the given tactic is applied on any proof obligation generated by the first command.

16.2 Syntax of annotated programs

16.2.1 Programs

The syntax of programs is given in figure 16.1. Basically, the programming language is a purely functional kernel with an addition of references and arrays on purely functional values. If you do not consider the logical assertions, the syntax coincide with Objective Caml syntax, except for elements of arrays which are written $t[i]$. In particular, the dereference of a mutable variable x is written $!x$ and assignment is written `:=` (for instance, the increment of the variable x will be written `x := !x + 1`). Actually, that syntax does not really matters, since it would be extracted later to real concrete syntax, in different programming languages.

Syntactic sugar.

- **Boolean expressions:**

  Boolean expressions appearing in programs (and in particular in if and while tests) are arbitrary programs of type `bool`. In order to make programs more readable, some syntactic sugar is provided for the usual logical connectives and the usual order relations over type `Z`, with the following syntax:

  $$\begin{array}{lcl} prog & ::= & prog \ \texttt{and} \ prog \\ & | & prog \ \texttt{or} \ prog \\ & | & \texttt{not} \ prog \\ & | & expression \ order\_relation \ expression \\ \\ order\_relation & ::= & > \mid >= \mid < \mid <= \mid = \mid <> \end{array}$$

  where the order relations have the strongest precedences, `not` has a stronger precedence than `and`, and `and` a stronger precedence than `or`.

  Order relations in other types, like `lt`, `le`, ... in type `nat`, should be explicited as described in the paragraph about *Boolean expressions*, page 246.

- **Arithmetical expressions:**

  Some syntactic sugar is provided for the usual arithmetic operator over type `Z`, with the following syntax:

  $$\begin{array}{lcl} prog & ::= & prog * prog \\ & | & prog + prog \\ & | & prog - prog \\ & | & - prog \end{array}$$

  where the unary operator − has the strongest precedence, and `*` a stronger precedence than + and −.

  Operations in other arithmetical types (such as type `nat`) must be explicitly written as applications, like `(plus a b)`, `(pred a)`, etc.

prog	::=	{ *predicate* } * *statement* [{ *predicate* }]
statement	::=	*expression*
	|	*identifier* `:=` *prog*
	|	*identifier* `[` *expression* `]` `:=` *prog*
	|	`let` *identifier* = `ref` *prog* `in` *prog*
	|	`if` *prog* `then` *prog* [`else` *prog*]
	|	`while` *prog* `do` *loop_annot* *block* `done`
	|	`begin` *block* `end`
	|	`let` *identifier* = *prog* `in` *prog*
	|	`fun` *binders* `->` *prog*
	|	`let` `rec` *identifier* *binder* `:` *value_type*
		`{` `variant` *wf_arg* `}` = *prog* [`in` *prog*]
	|	`(` *prog* *prog* `)`
expression	::=	*identifier*
	|	`!` *identifier*
	|	*identifier* `[` *expression* `]`
	|	*integer*
	|	`(` *expression* + `)`
block	::=	*block_statement* [`;` *block*]
block_statement	::=	*prog*
	|	`label` *identifier*
	|	`assert` `{` *predicate* `}`
binders	::=	`(` *identifier*, ..., *identifier* `:` *value_type* `)` +
loop_annot	::=	`{` `invariant` *predicate* `variant` *wf_arg* `}`
wf_arg	::=	*cic_term* [`for` *cic_term*]
predicate	::=	*cci_term* [`as` *identifier*]

Figure 16.1: Syntax of annotated programs

- `if` b `then` p is a shortcut for `if` b `then` p `else tt`, where `tt` is the constant of type `unit`;
- Values in type `Z` may be directly written as integers : 0,1,12546,...Negative integers are not recognized and must be written as `(Zinv` x`)`;
- Multiple application may be written $(f\ a_1\ \ldots\ a_n)$, which must be understood as left-associative i.e. as $(\ldots((f\ a_1)\ a_2)\ldots\ a_n)$.

Restrictions. You can notice some restrictions with respect to real ML programs:

1. Binders in functions (recursive or not) are explicitly typed, and the type of the result of a recursive function is also given. This is due to the lack of type inference.
2. Full expressions are not allowed on left-hand side of assignment, but only variables. Therefore, you can not write

   ```
   (if b then x else y) := 0
   ```

   But, in most cases, you can rewrite them into acceptable programs. For instance, the previous program may be rewritten into the following one:

   ```
   if b then x := 0 else y := 0
   ```

16.2.2 Typing

The types of annotated programs are split into two kinds: the types of *values* and the types of *computations*. Those two types families are recursively mutually defined with the following concrete syntax:

value_type	::=	*cic_term*
	|	*cic_term* `ref`
	|	`array` *cic_term* `of` *cic_term*
	|	`fun (` *x*`:`*value_type* `)+` *computation_type*
computation_type	::=	`returns` *identifier*`:`*value_type*
		[`reads` *identifier*, ..., *identifier*] [`writes` *identifier*, ..., *identifier*]
		[`pre` *predicate*] [`post` *predicate*]
		`end`
predicate	::=	*cic_term*

The typing is mostly the one of ML, without polymorphism. The user should notice that:

- Arrays are indexed over the type `Z` of binary integers (defined in the module `ZArith`);
- Expressions must have purely functional types, and can not be references or arrays (but, of course, you can pass mutables to functions as call-by-variable arguments);
- There is no partial application.

16.2.3 Specification

The specification part of programs is made of different kind of annotations, which are terms of sort Prop in the Calculus of Inductive Constructions.

Those annotations can refer to the values of the variables directly by their names. *There is no dereference operator "!" in annotations*. Annotations are read with the Coq parser, so you can use all the Coq syntax to write annotations. For instance, if x and y are references over integers (in type Z), you can write the following annotation

```
{ `0 < x <= x+y` }
```

In a post-condition, if necessary, you can refer to the value of the variable x *before* the evaluation with the notation $x@$. Actually, it is possible to refer to the value of a variable at any moment of the evaluation with the notation $x@l$, provided that l is a *label* previously inserted in your program (see below the paragraph about labels).

You have the possibility to give some names to the annotations, with the syntax

```
{ annotation as identifier }
```

and then the annotation will be given this name in the proof obligations. Otherwise, fresh names are given automatically, of the kind Post3, Pre12, Test4, etc. You are encouraged to give explicit names, in order not to have to modify your proof script when your proof obligations change (for instance, if you modify a part of the program).

Pre- and post-conditions

Each program, and each of its sub-programs, may be annotated by a pre-condition and/or a post-condition. The pre-condition is an annotation about the values of variables *before* the evaluation, and the post-condition is an annotation about the values of variables *before* and *after* the evaluation. Example:

```
{ `0 < x` } x := (Zplus !x !x) { `x@ < x` }
```

Moreover, you can assert some properties of the result of the evaluation in the post-condition, by referring to it through the name *result*. Example:

```
(Zs (Zplus !x !x)) { (Zodd result) }
```

Loops invariants and variants

Loop invariants and variants are introduced just after the do keyword, with the following syntax:

while B do
  { invariant I variant ϕ for R }
  $block$
done

The invariant I is an annotation about the values of variables when the loop is entered, since B has no side effects (B is a purely functional expression). Of course, I may refer to values of variables at any moment before the entering of the loop.

The variant ϕ must be given in order to establish the termination of the loop. The relation R must be a term of type $A \to A \to$ Prop, where ϕ is of type A. When R is not specified, then ϕ is assumed to be of type Z and the usual order relation on natural number is used.

Recursive functions

The termination of a recursive function is justified in the same way as loops, using a variant. This variant is introduced with the following syntax

$$\text{let rec } f\ (x_1 : V_1) \dots (x_n : V_n) : V\ \{\ \text{variant } \phi \text{ for } R\ \} = prog$$

and is interpreted as for loops. Of course, the variant may refer to the bound variables x_i. The specification of a recursive function is the one of its body, $prog$. Example:

$$\text{let rec } fact\ (x : Z) : Z\ \{\ \text{variant } x\} = \{\ x \geq 0\ \}\ \dots\ \{\ result = x!\ \}$$

Assertions inside blocks

Assertions may be inserted inside blocks, with the following syntax

$$\text{begin } block\_statements \dots;\ \text{assert } \{\ P\ \};\ block\_statements \dots\ \text{end}$$

The annotation P may refer to the values of variables at any labels known at this moment of evaluation.

Inserting labels in your program

In order to refer to the values of variables at any moment of evaluation of the program, you may put some *labels* inside your programs. Actually, it is only necessary to insert them inside blocks, since this is the only place where side effects can appear. The syntax to insert a label is the following:

$$\text{begin } block\_statements \dots;\ \text{label } L;\ block\_statements \dots\ \text{end}$$

Then it is possible to refer to the value of the variable x at step L with the notation $x@L$.

There is a special label 0 which is automatically inserted at the beginning of the program. Therefore, $x@0$ will always refer to the initial value of the variable x.

Notice that this mechanism allows the user to get rid of the so-called *auxiliary variables*, which are usually widely used in traditional frameworks to refer to previous values of variables.

Boolean expressions

As explained above, boolean expressions appearing in if and while tests are arbitrary programs of type `bool`. Actually, there is a little restriction: a test can not do some side effects. Usually, a test if annotated in such a way:

$$B\ \{\text{if } result \text{ then } T \text{ else } F\}$$

(The if then else construction in the annotation is the one of Coq !) Here T and F are the two propositions you want to get in the two branches of the test. If you do not annotate a test, then T and F automatically become $B = \text{true}$ and $B = \text{false}$, which is the usual annotation in Floyd-Hoare logic.

But you should take advantages of the fact that T and F may be arbitrary propositions, or you can even annotate B with any other kind of proposition (usually depending on $result$).

As explained in the paragraph about the syntax of boolean expression, some syntactic sugar is provided for usual order relations over type `Z`. When you write if $x < y\ \dots$ in your program, it is only a shortcut for if (`Z_lt_ge_bool` x y) ..., where `Z_lt_ge_bool` is the proof of $\forall x, y$: `Z`.$\exists b$: `bool`.(if b then $x < y$ else $x \geq y$) i.e. of a program returning a boolean with the expected post-condition. But you can use any other functional expression of such a type. In particular, the `Correctness` standard library comes with a bunch of decidability theorems on type `nat`:

$$\begin{array}{ll} zerop\_bool & : \forall n : \mathsf{nat}.\exists b : \texttt{bool}.\text{if } b \text{ then } n = 0 \text{ else } 0 < n \\ nat\_eq\_bool & : \forall n, m : \mathsf{nat}.\exists b : \texttt{bool}.\text{if } b \text{ then } n = m \text{ else } n \neq m \\ le\_lt\_bool & : \forall n, m : \mathsf{nat}.\exists b : \texttt{bool}.\text{if } b \text{ then } n \leq m \text{ else } m < n \\ lt\_le\_bool & : \forall n, m : \mathsf{nat}.\exists b : \texttt{bool}.\text{if } b \text{ then } n < m \text{ else } m \leq n \end{array}$$

which you can combine with the logical connectives.

It is often the case that you have a decidability theorem over some type, as for instance a theorem of decidability of equality over some type S:

$$S\_dec : (x, y : S)\{x = y\} + \{\neg x = y\}$$

Then you can build a test function corresponding to S_dec using the operator `bool_of_sumbool` provided with the `Prorgams` module, in such a way:

$$\texttt{Definition}\ S\_bool := [x, y : S](\texttt{bool\_of\_sumbool}\ ?\ ?\ (S\_dec\ x\ y))$$

Then you can use the test function S_bool in your programs, and you will get the hypothesis $x = y$ and $\neg x = y$ in the corresponding branches. Of course, you can do the same for any function returning some result in the constructive sum $\{A\} + \{B\}$.

16.3 Local and global variables

16.3.1 Global variables

You can declare a new global variable with the following command

`Global Variable` x `:` $value\_type$.

where x may be a reference, an array or a function. **Example:**

```
Parameter N : Z.
Global Variable x : Z ref.
Correctness foo { `x < N` } begin x := (Zmult 2 !x) end { `x < 2*N` }.
```

Each time you complete a correctness proof, the corresponding program is added to the programs environment. You can list the current programs environment with the command

`Show Programs.`

16.3.2 Local variables

Local variables are introduced with the following syntax

`let` x `=` `ref` e_1 `in` e_2

where the scope of x is exactly the program e_2. Notice that, as usual in ML, local variables must be initialized (here with e_1).

When specifying a program including local variables, you have to take care about their scopes. Indeed, the following two programs are not annotated in the same way:

- let $x = e_1$ in e_2 $\{ Q \}$
  The post-condition Q applies to e_2, and therefore x may appear in Q;
- (let $x = e_1$ in e_2) $\{ Q \}$
  The post-condition Q applies to the whole program, and therefore the local variable x may *not* appear in Q (it is beyond its scope).

16.4 Function call

Still following the syntax of ML, the function application is written $(f\ a_1\ \ldots\ a_n)$, where f is a function and the a_i's its arguments. Notice that f and the a_i's may be annotated programs themselves.

In the general case, f is a function already specified (either with `Global Variable` or with a proof of correctness) and has a pre-condition P_f and a post-condition Q_f.

As expected, a proof obligation is generated, which correspond to P_f applied to the values of the arguments, once they are evaluated.

Regarding the post-condition of f, there are different possible cases:

- either you did not annotate the function call, writing directly

$$(f\ a_1\ \ldots\ a_n)$$

  and then the post-condition of f is added automatically *if possible*: indeed, if some arguments of f make side-effects this is not always possible. In that case, you have to put a post-condition to the function call by yourself;

- or you annotated it with a post-condition, say Q:

$$(f\ a_1\ \ldots\ a_n)\ \{ Q \}$$

  then you will have to prove that Q holds under the hypothesis that the post-condition Q_f holds (where both are instantiated by the results of the evaluation of the a_i). Of course, if Q is exactly the post-condition of f then the corresponding proof obligation will be automatically discharged.

16.5 Libraries

The tactic comes with some libraries, useful to write programs and specifications. The first set of libraries is automatically loaded with the module `Correctness`. Among them, you can find the modules:

ProgWf : this module defines a family of relations Zwf on type `Z` by

$$(Zwf\ c) = \lambda x, y.c \leq x \wedge c \leq y \wedge x < y$$

and establishes that this relation is well-founded for all c (lemma `Zwf_well_founded`). This lemma is automatically used by the tactic `Correctness` when necessary. When no relation is given for the variant of a loop or a recursive function, then $(Zwf\ 0)$ is used *i.e.* the usual order relation on positive integers.

Arrays : this module defines an abstract type `array` for arrays, with the corresponding operations `new`, `access` and `store`. Access in a array t at index i may be written `#`t`[`i`]` in Coq, and in particular inside specifications. This module also provides some axioms to manipulate arrays expression, among which `store_def_1` and `store_def_2` allow you to simplify expressions of the kind `(access (store` t i v`)` j`)`.

Other useful modules, which are not automatically loaded, are the following:

Exchange : this module defines a predicate `(exchange` t t' i j`)` which means that elements of indexes i and j are swapped in arrays t and t', and other left unchanged. This modules also provides some lemmas to establish this property or conversely to get some consequences of this property.

Permut : this module defines the notion of permutation between two arrays, on a segment of the arrays (`sub_permut`) or on the whole array (`permut`). Permutations are inductively defined as the smallest equivalence relation containing the transpositions (defined in the module `Exchange`).

Sorted : this module defines the property for an array to be sorted, either on the whole array (`sorted_array`) or on a segment (`sub_sorted_array`). It also provides a few lemmas to establish this property.

16.6 Examples

16.6.1 Computation of X^n

As a first example, we prove the correctness of a program computing X^n using the following equations:

$$\left\{ \begin{array}{lcl} X^{2n} & = & (X^n)^2 \\ X^{2n+1} & = & X \times (X^n)^2 \end{array} \right.$$

If x and n are variables containing the input and y a variable that will contain the result (x^n), such a program may be the following one:

m := $!x$; y := 1 ;
while $!n \neq 0$ do
    if (odd $!n$) then y := $!y \times !m$;
    m := $!m \times !m$;
    n := $!n/2$
done

Specification part. Here we choose to use the binary integers of `ZArith`. The exponentiation X^n is defined, for $n \geq 0$, in the module `Zpower`:

```
Coq < Require ZArith.
```

```
Coq < Require Zpower.
```

In particular, the module `ZArith` loads a module `Zmisc` which contains the definitions of the predicate `Zeven` and `Zodd`, and the function `Zdiv2`. This module `ProgBool` also contains a test function `Zeven_odd_bool` of type $\forall n.\exists b.$if b then $(Zeven\ n)$ else $(Zodd\ n)$ derived from the proof `Zeven_odd_dec`, as explained in section 16.2.3:

Correctness part. Then we come to the correctness proof. We first import the Coq module `Correctness`:

```
Coq < Require Correctness.
```

Then we introduce all the variables needed by the program:

```
Coq < Parameter x : Z.
```

```
Coq < Global Variable n,m,y : Z ref.
```

At last, we can give the annotated program:

```
Coq < Correctness i_exp
Coq <   { `n >= 0` }
Coq <   begin
Coq <      m := x; y := 1;
Coq <      while !n > 0 do
Coq <        { invariant (Zpower x n@0)=(Zmult y (Zpower m n)) /\ `n >= 0`
Coq <          variant n }
Coq <        (if not (Zeven_odd_bool !n) then y := (Zmult !y !m))
Coq <           { (Zpower x n@0) = (Zmult y (Zpower m (Zdouble (Zdiv2 n)))) };
Coq <        m := (Zsquare !m);
Coq <        n := (Zdiv2 !n)
Coq <      done
Coq <    end
Coq <    { y=(Zpower x n@0) }
Coq < .
Toplevel input, characters 82-83
>      while !n > 0 do
>                    ^
Anomaly: Search error. Please report.
```

The proof obligations require some lemmas involving `Zpower` and `Zdiv2`. You can find the whole proof in the Coq standard library (see below). Let us make some quick remarks about this program and the way it was written:

- The name `n@0` is used to refer to the initial value of the variable `n`, as well inside the loop invariant as in the post-condition;
- Purely functional expressions are allowed anywhere in the program and they can use any purely informative Coq constants; That is why we can use `Zmult`, `Zsquare` and `Zdiv2` in the programs even if they are not other functions previously introduced as programs.

16.6.2 A recursive program

To give an example of a recursive program, let us rewrite the previous program into a recursive one. We obtain the following program:

let rec $exp\ x\ n$ =
  if $n = 0$ then
    1
  else
    let y = $(exp\ x\ (n/2))$ in
    if $(even\ n)$ then $y \times y$ else $x \times (y \times y)$

This recursive program, once it is annotated, is given to the tactic `Correctness`:

```
Coq < Correctness r_exp
Coq <   let rec exp (x:Z) (n:Z) : Z { variant n } =
Coq <     { `n >= 0` }
Coq <     (if n = 0 then
Coq <        1
Coq <      else
Coq <        let y = (exp x (Zdiv2 n)) in
Coq <        (if (Zeven_odd_bool n) then
Coq <           (Zmult y y)
Coq <         else
Coq <           (Zmult x (Zmult y y))) { result=(Zpower x n) }
Coq <     )
Coq <     { result=(Zpower x n) }
Coq < .
Toplevel input, characters 99-100
>     (if n = 0 then
>                  ^
Anomaly: Search error. Please report.
```

You can notice that the specification is simpler in the recursive case: we only have to give the pre- and post-conditions — which are the same as for the imperative version — but there is no annotation corresponding to the loop invariant. The other two annotations in the recursive program are added for the recursive call and for the test inside the let in construct (it can not be done automatically in general, so the user has to add it by himself).

16.6.3 Other examples

You will find some other examples with the distribution of the system Coq, in the sub-directory `contrib/correctness` of the Coq standard library. Those examples are mostly programs to compute the factorial and the exponentiation in various ways (on types `nat` or `Z`, in imperative way or recursively, with global variables or as functions, ...).

There are also some bigger correctness developments in the Coq contributions, which are available on the web page `coq.inria.fr/contribs`. for the moment, you can find:

- A proof of *insertion sort* by Nicolas Magaud, ENS Lyon;
- Proofs of *quicksort*, *heapsort* and *find* by the author.

These examples are fully detailed in [50, 48].

16.7 Bugs

- There is no discharge mechanism for programs; so you *cannot* do a program's proof inside a section (actually, you can do it, but your program will not exist anymore after having closed the section).

Surely there are still many bugs in this implementation. Please send bug reports to Jean-Christophe.Filliatre@lri.fr. Don't forget to send the version of Coq used (given by `coqtop -v`) and a script producing the bug.

Chapter 17

Extraction of programs in Objective Caml and Haskell

Jean-Christophe Filliâtre and Pierre Letouzey

The status of extraction is experimental.

We present here the Coq extraction commands, used to build certified and relatively efficient functional programs, extracting them from the proofs of their specifications. The functional languages available as output are currently Objective Caml and Haskell. In the following, "ML" will be used (abusively) to refer to any of the two.

Differences with old versions. The current extraction mechanism is new for version 7.0 of Coq. In particular, the F_ω toplevel used as an intermediate step between Coq and ML has been withdrawn. It is also not possible any more to import ML objects in this F_ω toplevel. The current mechanism also differs from the one in previous versions of Coq: there is no more an explicit toplevel for the language (formerly called Fml).

17.1 Generating ML code

The next two commands are meant to be used for rapid preview of extraction. They both display extracted term(s) inside Coq.

`Extraction` *term* `.`

Extracts one term in the Coq toplevel.

`Recursive Extraction` $qualid_1$ `...` $qualid_n$ `.`

Recursive extraction of all the globals $qualid_1 \dots qualid_n$ and all their dependencies in the Coq toplevel.

All the following commands produce real ML files. User can choose to produce one monolithic file or one file per Coq module.

`Extraction "file"` *qualid* $_1$ ... *qualid* $_n$.

Recursive extraction of all the globals *qualid* $_1$ ... *qualid* $_n$ and all their dependencies in one monolithic file *file*. Global and local identifiers are renamed according to the choosen ML language to fullfill its syntactic conventions, keeping original names as much as possible.

`Extraction Module` *ident*.

Extraction of the Coq module *ident* to an ML module *ident*`.ml`. In case of name clash, identifiers are here renamed using prefixes `coq_` or `Coq_` to ensure a session-independent renaming.

`Recursive Extraction Module` *ident*.

Extraction of the Coq module *ident* and all other modules *ident* depends on.

The list of globals *qualid* $_i$ does not need to be exhaustive: it is automatically completed into a complete and minimal environment. Extraction will fail if it encounters an informative axiom not realized (see section 17.2.3). A warning will be issued if it encounters an logical axiom, to remind user that inconsistant logical axioms may lead to incorrect or non-terminating extracted terms.

17.2 Extraction options

17.2.1 Setting the target language

The ability to fix target language is the first and more important of the extraction options. Default is Ocaml. Besides Haskell, another language called Toplevel is provided. It is a pseudo-Ocaml, with no renaming on global names: so names are printed as in Coq. This third language is available only at the Coq Toplevel.

`Extraction Language Ocaml.`

`Extraction Language Haskell.`

`Extraction Language Toplevel.`

17.2.2 Inlining and optimizations

Since Objective Caml is a strict language, the extracted code has to be optimized in order to be efficient (for instance, when using induction principles we do not want to compute all the recursive calls but only the needed ones). So the extraction mechanism provides an automatic optimization routine that will be called each time the user want to generate Ocaml programs. Essentially, it performs constants inlining and reductions. Therefore some constants may not appear in resulting monolithic Ocaml program (a warning is printed for each such constant). In the case of modular extraction, even if some inlining is done, the inlined constant are nevertheless printed, to ensure session-independent programs.

Concerning Haskell, such optimizations are less useful because of lazyness. We still make some optimizations, for example in order to produce more readable code.

All these optimizations are controled by the following Coq options:

`Set Extraction Optimize.`

`Unset Extraction Optimize.`

Default is Set. This control all optimizations made on the ML terms (mostly reduction of dummy beta/iota redexes, but also simplications on Cases, etc). Put this option to Unset if you want a ML term as close as possible to the Coq term.

`Set Extraction AutoInline.`

`Unset Extraction AutoInline.`

Default is Set, so by default, the extraction mechanism feels free to inline the bodies of some defined constants, according to some heuristics like size of bodies, useness of some arguments, etc. Those heuristics are not always perfect, you may want to disable this feature, do it by Unset.

`Extraction Inline` *qualid*$_1$ ... *qualid*$_n$.

`Extraction NoInline` *qualid*$_1$ ... *qualid*$_n$.

In addition to the automatic inline feature, you can now tell precisely to inline some more constants by the `Extraction Inline` command. Conversely, you can forbid the automatic inlining of some specific constants by the `Extraction NoInline` command. Those two commands enable a precise control of what is inlined and what is not.

`Print Extraction Inline.`

Prints the current state of the table recording the custom inlinings declared by the two previous commands.

`Reset Extraction Inline.`

Puts the table recording the custom inlinings back to empty.

Inlining and printing of a constant declaration. A user can explicitely asks a constant to be extracted by two means:

- by mentioning it on the extraction command line
- by extracting the whole Coq module of this constant.

In both cases, the declaration of this constant will be present in the produced file. But this same constant may or may not be inlined in the following terms, depending on the automatic/custom inlining mechanism.

For the constants non-explicitely required but needed for dependancy reasons, there are two cases:

- If an inlining decision is taken, wether automatically or not, all occurences of this constant are replaced by its extracted body, and this constant is not declared in the generated file.
- If no inlining decision is taken, the constant is normally declared in the produced file.

17.2.3 Realizing axioms

It is possible to assume some axioms while developing a proof. Since these axioms can be any kind of proposition or object type, they may perfectly well have some computational content. But 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. Of course, it is the responsability of the user to ensure that the ML terms given to realize the axioms do have the expected types.

The system actually provides a more general mechanism to specify ML terms even for defined constants, inductive types and constructors. For instance, the user may want to use the ML native boolean type instead of Coq one. The syntax is the following:

`Extract Constant` *qualid* `=>` *string* `.`

Give an ML extraction for the given constant. The *string* may be an identifier or a quoted string.

`Extract Inlined Constant` *qualid* `=>` *string* `.`

Same as the previous one, except that the given ML terms will be inlined everywhere instead of being declared via a let.

`Extract Inductive` *qualid* `=>` *string* `[` *string* `...`*string* `].`

Give an ML extraction for the given inductive type. You must specify extractions for the type itself (first *string*) and all its constructors (between square brackets). The ML extraction must be an ML recursive datatype.

Remarks:

1. The extraction of a module depending on axioms from another module will not fail. It is the responsability of the "extractor" of that other module to realize the given axioms.

2. Note that now, the `Extract Inlined Constant` command is sugar for an `Extract Constant` followed by a `Extraction Inline`. So be careful with `Reset Extraction Inline`.

Example: Typical examples are the following:

```
Coq < Extract Inductive unit => unit [ "()" ].

Coq < Extract Inductive bool => bool [ true false ].

Coq < Extract Inductive sumbool => bool [ true false ].
```

17.3 Differences between Coq and ML type systems

Due to differences between Coq and ML type systems, some extracted programs are not typable in ML. For example, Here are two kinds of problem that can occur:

- If some part of the program is *very* polymorphic, there may be no ML type for it. In that case the extraction to ML works all right but the generated code may be refused by the ML type-checker. A very well known example is the *distr-pair* function:

```
Definition dp := [A,B:Set][x:A][y:B][f:(C:Set)C->C](f A x,f B y).
```

In Ocaml, for instance, the extracted term is `let dp x y f = pair((f () x),(f () y))` and has type

```
dp : 'a -> 'a -> (unit -> 'a -> 'b) -> ('b,'b) prod
```

which is not its original type, but a restriction.

- Some definitions of Coq may have no counterpart in ML. This happens when there is a quantification over types inside the type of a constructor; for example:

```
Inductive anything : Set := dummy : (A:Set)A->anything.
```

which corresponds to the definition of ML dynamics.

It is still possible to run the programs by switching off the type-checker during compilation. Unless you misused the semantical attachment facilities you should never get any message like "segmentation fault" for which the extracted code would be to blame. For example, to bypass the Ocaml type-checker, we can use the function `Obj.magic` which gives the type `'a` to any object. Work is underway to generate those `Obj.magic` automatically.

We have to say, though, that in most "realistic" programs, these problems do not occur. For example all the programs of the library are accepted by Caml type-checker (see examples below).

17.4 Some examples

We present here two examples of extractions, taken from the Coq Standard Library. We choose Objective Caml as target language, but all can be done in the other dialects with slight modifications. We then indicate where to find other examples and tests of Extraction.

17.4.1 A detailed example: Euclidean division

The file `Euclid` contains the proof of Euclidean division (theorem `eucl_dev`). The natural numbers defined in the example files are unary integers defined by two constructors O and S:

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

This module contains a theorem `eucl_dev`, and its extracted term is of type

```
(b:nat)(gt b O)->(a:nat)(diveucl a b)
```

where `diveucl` is a type for the pair of the quotient and the modulo. We can now extract this program to Objective Caml:

```
Coq < Require Euclid.

Coq < Extraction Inline Wf_nat.gt_wf_rec Wf_nat.lt_wf_rec.

Coq < Recursive Extraction eucl_dev.
type sumbool =
  | Left
  | Right
type nat =
  | O
```

```
  | S of nat
type diveucl =
  | Divex of nat * nat
let rec minus n m =
  match n with
    | O -> O
    | S k -> (match m with
                | O -> S k
                | S l -> minus k l)
let rec le_lt_dec n m =
  match n with
    | O -> Left
    | S n0 -> (match m with
                 | O -> Right
                 | S n1 -> le_lt_dec n0 n1)
let le_gt_dec x x0 =
  le_lt_dec x x0
let rec eucl_dev b a =
  match le_gt_dec b a with
    | Left ->
        let Divex (x, x0) = eucl_dev b (minus a b) in Divex ((S x), x0)
    | Right -> Divex (O, a)
```

The inlining of `gt_wf_rec` and `lt_wf_rec` is not mandatory. It only enhances readability of extracted code. You can then copy-paste the output to a file `euclid.ml` or let Coq do it for you with the following command:

```
Coq < Extraction "euclid" eucl_dev.
```

Let us play the resulting program:

```
# #use "euclid.ml";;
type sumbool = Left | Right
type nat = O | S of nat
type diveucl = Divex of nat * nat
val minus : nat -> nat -> nat = <fun>
val le_lt_dec : nat -> nat -> sumbool = <fun>
val le_gt_dec : nat -> nat -> sumbool = <fun>
val eucl_dev : nat -> nat -> diveucl = <fun>
# eucl_dev (S (S O)) (S (S (S (S (S O)))));;
- : diveucl = Divex (S (S O), S O)
```

It is easier to test on Objective Caml integers:

```
# let rec i2n = function 0 -> O | n -> S (i2n (n-1));;
val i2n : int -> nat = <fun>
# let rec n2i = function O -> 0 | S p -> 1+(n2i p);;
val n2i : nat -> int = <fun>
# let div a b =
     let Divex (q,r) = eucl_dev (i2n b) (i2n a) in (n2i q, n2i r);;
div : int -> int -> int * int = <fun>
# div 173 15;;
- : int * int = 11, 8
```

17.4.2 Another detailed example: Heapsort

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

```
Coq < Require Heap.

Coq < Check treesort.
treesort
     : (A:Set; leA,eqA:(relation A))
         ((x,y:A){(leA x y)}+{(leA y x)})
         ->(eqA_dec:((x,y:A){(eqA x y)}+{~(eqA x y)}))
            ((x,y,z:A)(leA x y)->(leA y z)->(leA x z))
            ->(l:(list A))
               {m:(list A)| (sort leA m) & (permutation eqA_dec l m)}
```

Let's now extract this function:

```
Coq < Extraction NoInline list_to_heap.

Coq < Extraction "heapsort" treesort.
```

One more time, the `Extraction NoInline` directive is cosmetic. Without it, everything goes right, but `list_to_heap` is inlined inside `treesort`, producing a less readable term. Here is the produced file `heapsort.ml`:

```
type sumbool =
  | Left
  | Right

type nat =
  | O
  | S of nat

type 'a tree =
  | Tree_Leaf
  | Tree_Node of 'a * 'a tree * 'a tree

type 'a list =
  | Nil
  | Cons of 'a * 'a list

let rec merge leA_dec eqA_dec x x0 =
  match x with
    | Nil -> x0
    | Cons (a, l) ->
        let rec f = function
          | Nil -> Cons (a, l)
          | Cons (a0, l1) ->
              (match leA_dec a a0 with
```

```
                 | Left -> Cons (a,
                     (merge leA_dec eqA_dec l (Cons (a0, l1))))
                 | Right -> Cons (a0, (f l1)))
        in f x0

let rec heap_to_list leA_dec eqA_dec = function
  | Tree_Leaf -> Nil
  | Tree_Node (a, t, t0) -> Cons (a,
      (merge leA_dec eqA_dec (heap_to_list leA_dec eqA_dec t)
        (heap_to_list leA_dec eqA_dec t0)))

let rec insert leA_dec eqA_dec x x0 =
  match x with
    | Tree_Leaf -> Tree_Node (x0, Tree_Leaf, Tree_Leaf)
    | Tree_Node (a, t, t0) ->
        let h3 = fun x1 -> insert leA_dec eqA_dec t x1 in
        (match leA_dec a x0 with
           | Left -> Tree_Node (a, t0, (h3 x0))
           | Right -> Tree_Node (x0, t0, (h3 a)))

let rec list_to_heap leA_dec eqA_dec = function
  | Nil -> Tree_Leaf
  | Cons (a, l) ->
      insert leA_dec eqA_dec (list_to_heap leA_dec eqA_dec l) a

let treesort leA_dec eqA_dec l =
  heap_to_list leA_dec eqA_dec (list_to_heap leA_dec eqA_dec l)
```

Let's test it:

```
# #use "heapsort.ml";;
type sumbool = Left | Right
type nat = O | S of nat
type 'a tree = Tree_Leaf | Tree_Node of 'a * 'a tree * 'a tree
type 'a list = Nil | Cons of 'a * 'a list
val merge : ('a -> 'a -> sumbool) -> 'b -> 'a list -> 'a list -> 'a list =
  <fun>
val heap_to_list : ('a -> 'a -> sumbool) -> 'b -> 'a tree -> 'a list = <fun>
val insert : ('a -> 'a -> sumbool) -> 'b -> 'a tree -> 'a -> 'a tree = <fun>
val list_to_heap : ('a -> 'a -> sumbool) -> 'b -> 'a list -> 'a tree = <fun>
val treesort : ('a -> 'a -> sumbool) -> 'b -> 'a list -> 'a list = <fun>
```

One can remark that the argument of `treesort` corresponding to `eqAdec` is never used in the informative part of the terms, only in the logical parts. So the extracted `treesort` never use it, hence this `'b` argument. We will use `()` for this argument. Only remains the `leAdec` argument (of type `'a -> 'a -> sumbool`) to really provide.

```
# let leAdec x y = if x <= y then Left else Right;;
```

```
val leAdec : 'a -> 'a -> sumbool = <fun>
# let rec listn = function 0 -> Nil
                         | n -> Cons(Random.int 10000,listn (n-1));;
val listn : int -> int list = <fun>
# treesort leAdec () (listn 10);;
- : int list = Cons (136, Cons (760, Cons (1512, Cons (2776, Cons (3064,
Cons (4536, Cons (5768, Cons (7560, Cons (8856, Cons (8952, Nil))))))))))
```

Some tests on longer lists (10000 elements) show that the program is quite efficient for Caml code.

17.4.3 The Standard Library

As a test, we propose an automatic extraction of the Standard Library of Coq. In particular, we will find back the two previous examples, `Euclid` and `Heapsort`. Go to directory `contrib/extraction/test` of the sources of Coq, and run commands:

```
make tree; make
```

This will extract all Standard Library files and compile them. It is done via many `Extraction Module`, with some customization (see subdirectory `custom`).

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

```
http://www.lri.fr/~letouzey/extraction
```

Reals theory is normally not extracted, since it is an axiomatic development. We propose nonetheless a dummy realization of those axioms, to test, run:

```
make reals
```

This test works also with Haskell. In the same directory, run:

```
make tree; make -f Makefile.haskell
```

The haskell compiler currently used is `hbc`. Any other should also work, just adapt the `Makefile.haskell`. In particular `ghc` is known to work.

17.4.4 Extraction's horror museum

Some pathological examples of extraction are grouped in the file `contrib/extraction/test_extraction.v` of the sources of Coq.

17.4.5 Users' Contributions

Several of the Coq Users' Contributions use extraction to produce certified programs. In particular the following ones have an automatic extraction test (just run `make` in those directories):

- Bordeaux/Additions

- Bordeaux/EXCEPTIONS
- Bordeaux/SearchTrees
- Dyade/BDDS
- Lyon/CIRCUITS
- Lyon/FIRING-SQUAD
- Marseille/CIRCUITS
- Nancy/FOUnify
- Rocq/ARITH/Chinese
- Rocq/COC
- Rocq/GRAPHS
- Rocq/HIGMAN
- Sophia-Antipolis/Stalmarck
- Suresnes/BDD

Rocq/HIGMAN is a bit particular. The extracted code is normally not typable in ML due to an heavy use of impredicativity. So we realize one inductive type using an `Obj.magic` that artificially gives it the good type. After compilation this example runs nonetheless, thanks to the correction of the extraction (proof underway).

Chapter 18

The `Ring` tactic

Patrick Loiseleur and Samuel Boutin

This chapter presents the `Ring` tactic.

18.1 What does this tactic?

`Ring` does associative-commutative rewriting in ring and semi-ring structures. Assume you have two binary functions $\oplus$ and $\otimes$ that are associative and commutative, with $\oplus$ distributive on $\otimes$, and two constants 0 and 1 that are unities for $\oplus$ and $\otimes$. A *polynomial* is an expression built on variables $V_0, V_1, \ldots$ and constants by application of $\oplus$ and $\otimes$.

Let an *ordered product* be a product of variables $V_{i_1} \otimes \ldots \otimes V_{i_n}$ verifying $i_1 \leq i_2 \leq \ldots \leq i_n$. Let a *monomial* be the product of a constant (possibly equal to 1, in which case we omit it) and an ordered product. We can order the monomials by the lexicographic order on products of variables. Let a *canonical sum* be an ordered sum of monomials that are all different, i.e. each monomial in the sum is strictly less than the following monomial according to the lexicographic order. It is an easy theorem to show that every polynomial is equivalent (modulo the ring properties) to exactly one canonical sum. This canonical sum is called the *normal form* of the polynomial. So what does `Ring`? It normalizes polynomials over any ring or semi-ring structure. The basic utility of `Ring` is to simplify ring expressions, so that the user does not have to deal manually with the theorems of associativity and commutativity.

Examples:

1. In the ring of integers, the normal form of $x(3 + yx + 25(1 - z)) + zx$ is $28x + (-24)xz + xxy$.

2. For the classical propositional calculus (or the boolean rings) the normal form is what logicians call *disjunctive normal form*: every formula is equivalent to a disjunction of conjunctions of atoms. (Here $\oplus$ is $\vee$, $\otimes$ is $\wedge$, variables are atoms and the only constants are T and F)

18.2 The variables map

It is frequent to have an expression built with + and $\times$, but rarely on variables only. Let us associate a number to each subterm of a ring expression in the Gallina language. For example in the ring `nat`, consider the expression:

```
(plus (mult (plus (f (5)) x) x)
   (mult (if b then (4) else (f (3))) (2)))
```

As a ring expression, is has 3 subterms. Give each subterm a number in an arbitrary order:

0 $\mapsto$ `if b then (4) else (f (3))`
1 $\mapsto$ `(f (5))`
2 $\mapsto$ `x`

Then normalize the "abstract" polynomial

$$((V_1 \otimes V_2) \oplus V_2) \oplus (V_0 \otimes 2)$$

In our example the normal form is:

$$(2 \otimes V_0) \oplus (V_1 \otimes V_2) \oplus (V_2 \otimes V_2)$$

Then substitute the variables by their values in the variables map to get the concrete normal polynomial:

```
(plus (mult (2) (if b then (4) else (f (3))))
   (plus (mult (f (5)) x) (mult x x)))
```

18.3 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.

18.4 Concrete usage in Coq

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

```
Require Ring.
```

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

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

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

`Ring` $term_1 \ldots term_n$

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

Warning: `Ring` $term_1$`;` `Ring` $term_2$ is not equivalent to `Ring` $term_1$ $term_2$. 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.

Error messages:

1. `All terms must have the same type`

2. `Don't know what to do with this goal`

3. `No Declared Ring Theory for` *term*`.`
   `Use Add [Semi] Ring to declare it`
   That happens when all terms have the same type *term*, but there is no declared ring theory for this set. See below.

Variants:

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

   Error message: `This goal is not an equality`

18.5 Add a ring structure

It can be done in the Coqtoplevel (No ML file to edit and to link with Coq). First, `Ring` can handle two kinds of structure: rings and semi-rings. Semi-rings are like rings without an opposite to addition. Their precise specification (in Gallina) can be found in the file

```
contrib/ring/Ring_theory.v
```

The typical example of ring is `Z`, the typical example of semi-ring is `nat`.

The specification of a ring is divided in two parts: first the record of constants ($\oplus$, $\otimes$, 1, 0, $\ominus$) and then the theorems (associativity, commutativity, etc.).

```
Section Theory_of_semi_rings.

Variable A : Type.
Variable Aplus : A -> A -> A.
Variable Amult : A -> A -> A.
Variable Aone : A.
Variable Azero : A.
(* There is also a "weakly decidable" equality on A. That means
  that if (A_eq x y)=true then x=y but x=y can arise when
  (A_eq x y)=false. On an abstract ring the function [x,y:A]false
  is a good choice. The proof of A_eq_prop is in this case easy. *)
Variable Aeq : A -> A -> bool.

Record Semi_Ring_Theory : Prop :=
{ SR_plus_sym  : (n,m:A)[| n + m == m + n |];
  SR_plus_assoc : (n,m,p:A)[| n + (m + p) == (n + m) + p |];

  SR_mult_sym : (n,m:A)[| n*m == m*n |];
  SR_mult_assoc : (n,m,p:A)[| n*(m*p) == (n*m)*p |];
  SR_plus_zero_left :(n:A)[| 0 + n == n|];
```

```
  SR_mult_one_left : (n:A)[| 1*n == n |];
  SR_mult_zero_left : (n:A)[| 0*n == 0 |];
  SR_distr_left   : (n,m,p:A) [| (n + m)*p == n*p + m*p |];
  SR_plus_reg_left : (n,m,p:A)[| n + m == n + p |] -> m==p;
  SR_eq_prop : (x,y:A) (Is_true (Aeq x y)) -> x==y
}.

Section Theory_of_rings.

Variable A : Type.

Variable Aplus : A -> A -> A.
Variable Amult : A -> A -> A.
Variable Aone : A.
Variable Azero : A.
Variable Aopp : A -> A.
Variable Aeq : A -> A -> bool.


Record Ring_Theory : Prop :=
{ Th_plus_sym  : (n,m:A)[| n + m == m + n |];
  Th_plus_assoc : (n,m,p:A)[| n + (m + p) == (n + m) + p |];
  Th_mult_sym : (n,m:A)[| n*m == m*n |];
  Th_mult_assoc : (n,m,p:A)[| n*(m*p) == (n*m)*p |];
  Th_plus_zero_left :(n:A)[| 0 + n == n|];
  Th_mult_one_left : (n:A)[| 1*n == n |];
  Th_opp_def : (n:A) [| n + (-n) == 0 |];
  Th_distr_left   : (n,m,p:A) [| (n + m)*p == n*p + m*p |];
  Th_eq_prop : (x,y:A) (Is_true (Aeq x y)) -> x==y
}.
```

To define a ring structure on A, you must provide an addition, a multiplication, an opposite function and two unities 0 and 1.

You must then prove all theorems that make (A,Aplus,Amult,Aone,Azero,Aeq) a ring structure, and pack them with the `Build_Ring_Theory` constructor.

Finally to register a ring the syntax is:

> `Add Ring` *A Aplus Amult Aone Azero Ainv Aeq T* `[` *c1 ... cn* `]`.

where *A* is a term of type `Set`, *Aplus* is a term of type `A->A->A`, *Amult* is a term of type `A->A->A`, *Aone* is a term of type `A`, *Azero* is a term of type `A`, *Ainv* is a term of type `A->A`, *Aeq* is a term of type `A->bool`, *T* is a term of type `(Ring_Theory` *A Aplus Amult Aone Azero Ainv Aeq*`)`. The arguments *c1 ... cn*, are the names of constructors which define closed terms: a subterm will be considered as a constant if it is either one of the terms *c1 ... cn* or the application of one of these terms to closed terms. For `nat`, the given constructors are `S` and `O`, and the closed terms are `O`, `(S O)`, `(S (S O))`, `...`

Variants:

1. `Add Semi Ring` *A Aplus Amult Aone Azero Aeq T* `[` *c1 ... cn* `]`.
   There are two differences with the `Add Ring` command: there is no inverse function and the term *T* must be of type `(Semi_Ring_Theory` *A Aplus Amult Aone Azero Aeq*`)`.

2. `Add Abstract Ring` *A Aplus Amult Aone Azero Ainv Aeq T* `.`
   This command should be used for when the operations of rings are not computable; for example the real numbers of `theories/REALS/`. Here $0 + 1$ is not beta-reduced to 1 but you still may want to *rewrite* it to 1 using the ring axioms. The argument `Aeq` is not used; a good choice for that function is `[x:A]false`.

3. `Add Abstract Semi Ring` *A Aplus Amult Aone Azero Aeq T* `.`

Error messages:

1. `Not a valid (semi)ring theory.`
   That happens when the typing condition does not hold.

Currently, the hypothesis is made than no more than one ring structure may be declared for a given type in `Set` or `Type`. This allows automatic detection of the theory used to achieve the normalization. On popular demand, we can change that and allow several ring structures on the same set.

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

The typical example of ring is `Z`, and the typical example of semi-ring is `nat`. Another ring structure is defined on the booleans.

Warning: Only the ring of booleans is loaded by default with the `Ring` module. To load the ring structure for `nat`, load the module `ArithRing`, and for `Z`, load the module `ZArithRing`.

18.6 How does it work?

The code of `Ring` a good example of tactic written using *reflection* (or *internalization*, it is synonymous). What is reflection? Basically, it is writing Coq tactics in Coq, rather than in Objective Caml. From the philosophical point of view, it is using the ability of the Calculus of Constructions to speak and reason about itself. For the `Ring` tactic we used Coq as a programming language and also as a proof environment to build a tactic and to prove it correctness.

The interested reader is strongly advised to have a look at the file `Ring_normalize.v`. Here a type for polynomials is defined:

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

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

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

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

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

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

So now, what is the scheme for a normalization proof? Let `p` be the polynomial expression that the user wants to normalize. First a little piece of ML code guesses the type of `p`, the ring theory `T` to use, an abstract polynomial `ap` and a variables map `v` such that `p` is $\beta\delta\iota$-equivalent to `(interp v ap)`. Then we replace it by `(interp v (polynomial_simplify ap))`, using the main correctness theorem and we reduce it to a concrete expression `p'`, which is the concrete normal form of `p`. This is summarized in this diagram:

$$
\begin{array}{ccl}
\texttt{p} & \rightarrow_{\beta\delta\iota} & \texttt{(interp v ap)} \\
 & & =_{\text{(by the main correctness theorem)}} \\
\texttt{p'} & \leftarrow_{\beta\delta\iota} & \texttt{(interp v (polynomial\_simplify ap))}
\end{array}
$$

The user do not see the right part of the diagram. From outside, the tactic behaves like a $\beta\delta\iota$ simplification extended with AC rewriting rules. Basically, the proof is only the application of the main correctness theorem to well-chosen arguments.

18.7 History of **Ring**

First Samuel Boutin designed the tactic `ACDSimpl`. This tactic did lot of rewriting. But the proofs terms generated by rewriting were too big for Coq's type-checker. Let us see why:

```
Coq < Goal (x,y,z:Z)`x + 3 + y + y*z = x + 3 + y + z*y`.
1 subgoal

  ============================
   (x,y,z:Z)`x+3+y+y*z = x+3+y+z*y`

Coq < Intros; Rewrite (Zmult_sym y z); Reflexivity.

Coq < Save toto.

Coq < Print toto.
toto =
[x,y,z:Z]
 (eq_ind_r Z `z*y` [z0:Z]`x+3+y+z0 = x+3+y+z*y`
   (refl_equal Z `x+3+y+z*y`) `y*z` (Zmult_sym y z))
     : (x,y,z:Z)`x+3+y+y*z = x+3+y+z*y`
```

At each step of rewriting, the whole context is duplicated in the proof term. Then, a tactic that does hundreds of rewriting generates huge proof terms. Since `ACDSimpl` was too slow, Samuel Boutin rewrote it using reflection (see his article in TACS'97 [14]). Later, the stuff was rewritten by Patrick Loiseleur: the new tactic does not any more require `ACDSimpl` to compile and it makes use of $\beta\delta\iota$-reduction not only to replace the rewriting steps, but also to achieve the interleaving of computation and reasoning (see 18.8). He also wrote a few ML code for the `Add Ring` command, that allow to register new rings dynamically.

Proofs terms generated by `Ring` are quite small, they are linear in the number of $\oplus$ and $\otimes$ operations in the normalized terms. Type-checking those terms requires some time because it makes a large use of the conversion rule, but memory requirements are much smaller.

18.8 Discussion

Efficiency is not the only motivation to use reflection here. `Ring` also deals with constants, it rewrites for example the expression $34 + 2 * x - x + 12$ to the expected result $x + 46$. For the tactic `ACDSimpl`, the only constants were 0 and 1. So the expression $34 + 2 * (x - 1) + 12$ is interpreted as $V_0 \oplus V_1 \otimes (V_2 \ominus 1) \oplus V_3$, with the variables mapping $\{V_0 \mapsto 34; V_1 \mapsto 2; V_2 \mapsto x; V_3 \mapsto 12\}$. Then it is rewritten to $34 - x + 2 * x + 12$, very far from the expected result. Here rewriting is not sufficient: you have to do some kind of reduction (some kind of *computation*) to achieve the normalization.

The tactic `Ring` is not only faster than a classical one: using reflection, we get for free integration of computation and reasoning that would be very complex to implement in the classic fashion.

Is it the ultimate way to write tactics? The answer is: yes and no. The `Ring` tactic uses intensively the conversion rule of CIC, that is replaces proof by computation the most as it is possible. It can be useful in all situations where a classical tactic generates huge proof terms. Symbolic Processing and Tautologies are in that case. But there are also tactics like `Auto` or `Linear`: that do many complex computations, using side-effects and backtracking, and generate a small proof term. Clearly, it would be a non-sense to replace them by tactics using reflection.

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

Another brilliant idea from Benjamin Werner: reflection could be used to couple a external tool (a rewriting program or a model checker) with Coq. We define (in Coq) a type of terms, a type of *traces*, and prove a correction theorem that states that *replaying traces* is safe w.r.t some interpretation. Then we let the external tool do every computation (using side-effects, backtracking, exception, or others features that are not available in pure lambda calculus) to produce the trace: now we replay the trace in Coq, and apply the correction lemma. So internalization seems to be the best way to import ... external proofs!

Chapter 19

The Setoid_replace tactic

Clément Renard

This chapter presents the `Setoid_replace` tactic.

19.1 Description of Setoid_replace

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

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

19.2 Adding new setoid or morphisms

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

```
Require Setoid.
```

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

The specification of a setoid can be found in the file

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

It looks like :

```
Section Setoid.

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

Record Setoid_Theory : Prop :=
{ Seq_refl : (x:A) (Aeq x x);
  Seq_sym : (x,y:A) (Aeq x y) -> (Aeq y x);
  Seq_trans : (x,y,z:A) (Aeq x y) -> (Aeq y z) -> (Aeq x z)
}.
```

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

Finally to register a setoid the syntax is:

`Add Setoid` *A Aeq ST*

where *Aeq* is a term of type `A->A->Prop` and *ST* is a term of type `(Setoid_Theory` *A Aeq*`)`.

Error messages:

1. `Not a valid setoid theory.`
   That happens when the typing condition does not hold.

2. `A Setoid Theory is already declared for` *A*.
   That happens when you try to declare a second setoid theory for the same type.

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

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

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

19.3 Adding new morphisms

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

The syntax is the following :

`Add Morphism` *f* : *ident*

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

Error messages:

1. `The term` *term* `is already declared as a morphism`

2. `The term` *term* `is not a product`

3. `The term` *term* `should not be a dependent product`

The compatibility lemma genereted depends on the setoids already declared.

19.4 The tactic itself

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

`Setoid_replace` $term_1$ with $term_2$

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

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

`Setoid_rewrite` *term*

Variants:

1. `Setoid_rewrite ->` *term*
2. `Setoid_rewrite <-` *term*

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

Bibliography

[1] Ph. Audebaud. Partial Objects in the Calculus of Constructions. In *Proceedings of the sixth Conf. on Logic in Computer Science.* IEEE, 1991.

[2] Ph. Audebaud. CC+ : an extension of the Calculus of Constructions with fixpoints. In B. Nordström and K. Petersson and G. Plotkin, editor, *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages pp 21–34, 1992. Also Research Report LIP-ENS-Lyon.

[3] Ph. Audebaud. *Extension du Calcul des Constructions par Points fixes*. PhD thesis, Université Bordeaux I, 1992.

[4] L. Augustsson. Compiling Pattern Matching. In *Conference Functional Programming and Computer Architecture*, 1985.

[5] H. Barendregt. Lambda Calculi with Types. Technical Report 91-19, Catholic University Nijmegen, 1991. In Handbook of Logic in Computer Science, Vol II.

[6] H. Barendregt and T. Nipkow, editors. *Types for Proofs and Programs*, volume 806 of *LNCS*. Springer-Verlag, 1994.

[7] H.P. Barendregt. *The Lambda Calculus its Syntax and Semantics*. North-Holland, 1981.

[8] J.L. Bates and R.L. Constable. Proofs as Programs. *ACM transactions on Programming Languages and Systems*, 7, 1985.

[9] M.J. Beeson. *Foundations of Constructive Mathematics, Metamathematical Studies*. Springer-Verlag, 1985.

[10] G. Bellin and J. Ketonen. A decision procedure revisited : Notes on direct logic, linear logic and its implementation. *Theoretical Computer Science*, 95:115–142, 1992.

[11] E. Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, 1967.

[12] S. Boutin. Certification d'un compilateur ML en Coq. Master's thesis, Université Paris 7, September 1992.

[13] S. Boutin. *Réflexions sur les quotients*. thèse d'université, Paris 7, April 1997.

[14] S. Boutin. Using reflection to build efficient and certified decision procedure s. In Martin Abadi and Takahashi Ito, editors, *TACS'97*, volume 1281. LNCS, Springer-Verlag, 1997.

[15] R.S. Boyer and J.S. Moore. *A computational logic*. ACM Monograph. Academic Press, 1979.

[16] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[17] Th. Coquand. *Une Théorie des Constructions*. PhD thesis, Université Paris 7, January 1985.

[18] Th. Coquand. An Analysis of Girard's Paradox. In *Symposium on Logic in Computer Science*, Cambridge, MA, 1986. IEEE Computer Society Press.

[19] Th. Coquand. Metamathematical Investigations of a Calculus of Constructions. In P. Oddifredi, editor, *Logic and Computer Science*. Academic Press, 1990. INRIA Research Report 1088, also in [52].

[20] Th. Coquand. Pattern Matching with Dependent Types. In Nordström et al. [85].

[21] Th. Coquand. Infinite Objects in Type Theory. In Barendregt and Nipkow [6].

[22] Th. Coquand and G. Huet. Constructions : A Higher Order Proof System for Mechanizing Mathematics. In *EUROCAL'85*, volume 203 of *LNCS*, Linz, 1985. Springer-Verlag.

[23] Th. Coquand and G. Huet. Concepts Mathématiques et Informatiques formalisés dans le Calcul des Constructions. In The Paris Logic Group, editor, *Logic Colloquium'85*. North-Holland, 1987.

[24] Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.

[25] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *LNCS*. Springer-Verlag, 1990.

[26] J. Courant. Explicitation de preuves par récurrence implicite. Master's thesis, DEA d'Informatique, ENS Lyon, September 1994.

[27] N.J. de Bruijn. Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indag. Math.*, 34, 1972.

[28] N.J. de Bruijn. A survey of the project Automath. In J.P. Seldin and J.R. Hindley, editors, *to H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980.

[29] D. de Rauglaudre. Camlp4 version 1.07.2. In Camlp4 distribution, 1998.

[30] D. Delahaye. Information Retrieval in a Coq Proof Library using Type Isomorphisms. In *Proceedings of TYPES'99, Lökeberg*. Springer-Verlag LNCS, 1999. ftp://ftp.inria.fr/INRIA/Projects/coq/David.Delahaye/papers/TYPES99-SIsos.ps.gz.

[31] D. Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955, pages 85–95. Springer-Verlag LNCS/LNAI, November 2000. ftp://ftp.inria.fr/INRIA/Projects/coq/David.Delahaye/papers/LPAR2000-ltac.ps.gz.

[32] D. Delahaye and M. Mayero. `Field`: une procédure de décision pour les nombres réels en Coq. In *Journées Francophones des Langages Applicatifs, Pontarlier*. INRIA, Janvier 2001. ftp://ftp.inria.fr/INRIA/Projects/coq/David.Delahaye/papers/JFLA2000-Field.ps.gz.

[33] 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.

[34] G. Dowek. Naming and Scoping in a Mathematical Vernacular. Research Report 1283, INRIA, 1990.

[35] G. Dowek. A Second Order Pattern Matching Algorithm in the Cube of Typed λ-calculi. In *Proceedings of Mathematical Foundation of Computer Science*, volume 520 of *LNCS*, pages 151–160. Springer-Verlag, 1991. Also INRIA Research Report.

[36] G. Dowek. *Démonstration automatique dans le Calcul des Constructions*. PhD thesis, Université Paris 7, December 1991.

[37] G. Dowek. L'Indécidabilité du Filtrage du Troisième Ordre dans les Calculs avec Types Dépendants ou Constructeurs de Types. *Compte Rendu de l'Académie des Sciences*, I, 312(12):951–956, 1991. (The undecidability of Third Order Pattern Matching in Calculi with Dependent Types or Type Constructors).

[38] G. Dowek. The Undecidability of Pattern Matching in Calculi where Primitive Recursive Functions are Representable. To appear in Theoretical Computer Science, 1992.

[39] G. Dowek. A Complete Proof Synthesis Method for the Cube of Type Systems. *Journal Logic Computation*, 3(3):287–315, June 1993.

[40] G. Dowek. Third order matching is decidable. *Annals of Pure and Applied Logic*, 69:135–155, 1994.

[41] G. Dowek. Lambda-calculus, Combinators and the Comprehension Schema. In *Proceedings of the second international conference on typed lambda calculus and applications*, 1995.

[42] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq Proof Assistant User's Guide Version 5.8. Technical Report 154, INRIA, May 1993.

[43] P. Dybjer. Inductive sets and families in Martin-Löf's Type Theory and their set-theoretic semantics : An inversion principle for Martin-Löf's type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, volume 14, pages 59–79. Cambridge University Press, 1991.

[44] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3), September 1992.

[45] J.-C. Filliâtre. Une procédure de décision pour le Calcul des Prédicats Direct. Etude et implémentation dans le système Coq. Master's thesis, DEA d'Informatique, ENS Lyon, September 1994.

[46] J.-C. Filliâtre. A decision procedure for Direct Predicate Calculus. Research report 96–25, LIP-ENS-Lyon, 1995.

[47] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université Paris-Sud, July 1999. `http://www.lri.fr/~filliatr/ftp/publis/these.ps.gz`.

[48] J.-C. Filliâtre. Formal Proof of a Program: Find. Submitted to *Science of Computer Programming*, January 2000.

[49] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. To appear in the Journal of Functional Programming. [English translation of [47]], February 2000.

[50] J.-C. Filliâtre and N. Magaud. Certification of sorting algorithms in the system Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999. `http://www.lri.fr/~filliatr/ftp/publis/Filliatre-Magaud.ps.gz`.

[51] E. Fleury. Implantation des algorithmes de Floyd et de Dijkstra dans le Calcul des Constructions. Rapport de Stage, July 1990.

[52] Projet Formel. The Calculus of Constructions. Documentation and user's guide, Version 4.10. Technical Report 110, INRIA, 1989.

[53] Jean-Baptiste-Joseph Fourier. *Fourier's method to solve linear inequations/equations systems.* Gauthier-Villars, 1890.

[54] E. Giménez. Codifying guarded definitions with recursive schemes. In *Types'94 : Types for Proofs and Programs*, volume 996 of *LNCS*. Springer-Verlag, 1994. Extended version in LIP research report 95-07, ENS Lyon.

[55] 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 LNCS, pages 135–152. Springer-Verlag, 1995.

[56] E. Giménez. A tutorial on recursive types in coq. Technical report, INRIA, March 1998.

[57] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the 2nd Scandinavian Logic Symposium*. North-Holland, 1970.

[58] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972.

[59] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989.

[60] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Centre,, 1995.

[61] D. Hirschkoff. Ecriture d'une tactique arithmétique pour le système Coq. Master's thesis, DEA IARFA, Ecole des Ponts et Chaussées, Paris, September 1994.

[62] W.A. Howard. The formulae-as-types notion of constructions. In J.P. Seldin and J.R. Hindley, editors, *to H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980. Unpublished 1969 Manuscript.

[63] G. Huet. Induction principles formalized in the Calculus of Constructions. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*. Elsevier Science, 1988. Also in Proceedings of TAPSOFT87, LNCS 249, Springer-Verlag, 1987, pp 276–286.

[64] G. Huet, editor. *Logical Foundations of Functional Programming*. The UT Year of Programming Series. Addison-Wesley, 1989.

[65] G. Huet. The Constructive Engine. In R. Narasimhan, editor, *A perspective in Theoretical Computer Science. Commemorative Volume for Gift Siromoney*. World Scientific Publishing, 1989. Also in [52].

[66] G. Huet. The Gallina Specification Language : A case study. In *Proceedings of 12th FST/TCS Conference, New Delhi*, volume 652 of *LNCS*, pages 229–240. Springer Verlag, 1992.

[67] G. Huet. Residual theory in λ-calculus: a formal development. *J. Functional Programming*, 4,3:371–394, 1994.

[68] G. Huet and J.-J. Lévy. Call by need computations in non-ambigous linear term rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*. The MIT press, 1991. Also research report 359, INRIA, 1979.

[69] G. Huet and G. Plotkin, editors. *Logical Frameworks*. Cambridge University Press, 1991.

[70] G. Huet and G. Plotkin, editors. *Logical Environments*. Cambridge University Press, 1992.

[71] J. Ketonen and R. Weyhrauch. A decidable fragment of Predicate Calculus. *Theoretical Computer Science*, 32:297–307, 1984.

[72] S.C. Kleene. *Introduction to Metamathematics*. Bibliotheca Mathematica. North-Holland, 1952.

[73] J.-L. Krivine. *Lambda-calcul types et modèles*. Etudes et recherche en informatique. Masson, 1990.

[74] A. Laville. Comparison of priority rules in pattern matching and term rewriting. *Journal of Symbolic Computation*, 11:321–347, 1991.

[75] F. Leclerc and C. Paulin-Mohring. Programming with Streams in Coq. A case study : The Sieve of Eratosthenes. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs, Types' 93*, volume 806 of *LNCS*. Springer-Verlag, 1994.

[76] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[77] L.Puel and A. Suárez. Compiling Pattern Matching by Term Decomposition. In *Conference Lisp and Functional Programming*, ACM. Springer-Verlag, 1990.

[78] P. Manoury. A User's Friendly Syntax to Define Recursive Functions as Typed $\lambda-$Terms. In *Types for Proofs and Programs, TYPES'94*, volume 996 of *LNCS*, June 1994.

[79] P. Manoury and M. Simonot. Automatizing termination proof of recursively defined function. *TCS*, To appear.

[80] L. Maranget. Two Techniques for Compiling Lazy Pattern Matching. Technical Report 2385, INRIA, 1994.

[81] C. Muñoz. Démonstration automatique dans la logique propositionnelle intuitionniste. Master's thesis, DEA d'Informatique Fondamentale, Université Paris 7, September 1994.

[82] C. Muñoz. *Un calcul de substitutions pour la représentation de preuves partielles en théorie de types*. Thèse de doctorat, Université Paris 7, 1997. Version en anglais disponible comme rapport de recherche INRIA RR-3309.

[83] B. Nordström. Terminating general recursion. *BIT*, 28, 1988.

[84] B. Nordström, K. Peterson, and J. Smith. *Programming in Martin-Löf's Type Theory*. International Series of Monographs on Computer Science. Oxford Science Publications, 1990.

[85] B. Nordström, K. Petersson, and G. Plotkin, editors. *Proceedings of the 1992 Workshop on Types for Proofs and Programs*. Available by ftp at site ftp.inria.fr, 1992.

[86] P. Odifreddi, editor. *Logic and Computer Science*. Academic Press, 1990.

[87] P. Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, 1984.

[88] C. Parent. Developing certified programs in the system Coq- The Program tactic. Technical Report 93-29, Ecole Normale Supérieure de Lyon, October 1993. Also in [6].

[89] C. Parent. *Synthèse de preuves de programmes dans le Calcul des Constructions Inductives*. PhD thesis, Ecole Normale Supérieure de Lyon, 1995.

[90] 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.

[91] M. Parigot. Recursive Programming with Proofs. *Theoretical Computer Science*, 94(2):335–356, 1992.

[92] M. Parigot, P. Manoury, and M. Simonot. ProPre : A Programming language with proofs. In A. Voronkov, editor, *Logic Programming and automated reasoning*, number 624 in LNCS, St. Petersburg, Russia, July 1992. Springer-Verlag.

[93] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM.

[94] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Université Paris 7, January 1989.

[95] 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.

[96] C. Paulin-Mohring. *Le système Coq. Thèse d'habilitation*. ENS Lyon, January 1997.

[97] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[98] K.V. Prasad. Programming with broadcasts. In *Proceedings of CONCUR'93*, volume 715 of *LNCS*. Springer-Verlag, 1993.

[99] J. Rouyer. Développement de l'Algorithme d'Unification dans le Calcul des Constructions. Technical Report 1795, INRIA, November 1992.

[100] A. Saïbi. Axiomatization of a lambda-calculus with explicit-substitutions in the Coq System. Technical Report 2345, INRIA, December 1994.

[101] H. Saidi. Résolution d'équations dans le système t de gödel. Master's thesis, DEA d'Informatique Fondamentale, Université Paris 7, September 1994.

[102] D. Terrasse. Traduction de TYPOL en COQ. Application à Mini ML. Master's thesis, IARFA, September 1992.

[103] L. Théry, Y. Bertot, and G. Kahn. Real theorem provers deserve real user-interfaces. Research Report 1684, INRIA Sophia, May 1992.

[104] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, an introduction*. Studies in Logic and the foundations of Mathematics, volumes 121 and 123. North-Holland, 1988.

[105] P. Wadler. Efficient compilation of pattern matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[106] P. Weis and X. Leroy. *Le langage Caml*. InterEditions, 1993.

[107] B. Werner. *Une théorie des constructions inductives*. Thèse de doctorat, Université Paris 7, 1994.

Global Index

Tactics Index

Vernacular Commands Index

Index of Error Messages