Translation from Coq V7 to V8

The Coq Development Team

January 2, 2004

1 Introduction

Coq version 8.0 is a major version and carries major changes: the concrete syntax was redesigned almost from scratch, and many notions of the libraries were renamed for uniformisation purposes. We felt that these changes could discourage users with large theories from switching to the new version.

The goal of this document is to introduce these changes on simple examples (mainly the syntactic changes), and describe the automated tools to help moving to V8.0. Essentially, it consists of a translator that takes as input a Coq source file in old syntax and produces a file in new syntax and adapted to the new standard library. The main extra features of this translator is that it keeps comments, even those within expressions[1].

The document is organised as follows: first section describes the new syntax on simple examples. It is very translation-oriented. This should give users of older versions the flavour of the new syntax, and allow them to make translation manually on small examples. Section 3 explains how the translation process can be automatised for the most part (the boring one: applying similar changes over thousands of lines of code). We strongly advise users to follow these indications, in order to avoid many potential complications of the translation process.

2 The new syntax on examples

The goal of this section is to introduce to the new syntax of Coq on simple examples, rather than just giving the new grammar. It is strongly recommended to read first the definition of the new syntax (in the reference manual), but this document should also be useful for the eager user who wants to start with the new syntax quickly.

The toplevel has an option `-translate` which allows to interactively translate commands. This toplevel translator accepts a command, prints the translation on standard output (after a `New syntax:` balise), executes the command, and waits for another command. The only requirements is that they should be syntactically correct, but they do not have to be well-typed.

This interactive translator proved to be useful in two main usages. First as a “debugger” of the translation. Before the translation, it may help in spotting possible conflicts between the new syntax and user notations. Or when the translation fails for some reason, it makes it easy to find the exact reason why it failed and make attempts in fixing the problem.

The second usage of the translator is when trying to make the first proofs in new syntax. Well trained users will automatically think their scripts in old syntax and might waste much time (and the intuition of the proof) if they have to search the translation in a document. Running a translator in the background will allow the user to instantly have the answer.

[1] The position of those comment might differ slightly since there is no exact matching of positions between old and new syntax.

The rest of this section is a description of all the aspects of the syntax that changed and how they were translated. All the examples below can be tested by entering the V7 commands in the toplevel translator.

2.1 Changes in lexical conventions w.r.t. V7

2.1.1 Identifiers

The lexical conventions changed: _ is not a regular identifier anymore. It is used in terms as a placeholder for subterms to be inferred at type-checking, and in patterns as a non-binding variable.

Furthermore, only letters (unicode letters), digits, single quotes and _ are allowed after the first character.

2.1.2 Quoted string

Quoted strings are used typically to give a filename (which may not be a regular identifier). As before they are written between double quotes ("). Unlike for V7, there is no escape character: characters are written normaly but the double quote which is doubled.

V7	V8
`"abcd\\efg"`	`"abcd\efg"`
`"abcd\"efg"`	`"abcd""efg"`

2.2 Main changes in terms w.r.t. V7

2.2.1 Precedence of application

In the new syntax, parentheses are not really part of the syntax of application. The precedence of application (10) is tighter than all prefix and infix notations. It makes it possible to remove parentheses in many contexts.

V7	V8
`(A x)->(f x)=(g y)`	`A x -> f x = g y`
`(f [x]x)`	`f (fun x => x)`

2.2.2 Arithmetics and scopes

The specialized notation for **Z** and **R** (introduced by symbols **‘** and **“**) have disappeared. They have been replaced by the general notion of scope.

type	scope name	delimiter
types	type_scope	**type**
bool	bool_scope	
nat	nat_scope	**nat**
Z	Z_scope	**Z**
R	R_scope	**R**
positive	positive_scope	**P**

In order to use notations of arithmetics on **Z**, its scope must be opened with command `Open Scope Z_scope.` Another possibility is using the scope change notation (**%**). The latter notation is to be used when notations of several scopes appear in the same expression.

In examples below, scope changes are not needed if the appropriate scope has been opened. Scope nat_scope is opened in the initial state of Coq.

V7	V8	
‘0+x=x+0‘	0+x=x+0	Z_scope
“0 + [if b then “1“ else “2“]“	0 + if b then 1 else 2	R_scope
(0)	0	nat_scope

Below is a table that tells which notation is available in which scope. The relative precedences and associativity of operators is the same as in usual mathematics. See the reference manual for more details. However, it is important to remember that unlike V7, the type operators for product and sum are left associative, in order not to clash with arithmetic operators.

scope	notations
nat_scope	+ - * < <= > >=
Z_scope	+ - * / mod < <= > >= ?=
R_scope	+ - * / < <= > >=
type_scope	* +
bool_scope	&& || -
list_scope	:: ++

2.2.3 Notation for implicit arguments

The explicitation of arguments is closer to the *bindings* notation in tactics. Argument positions follow the argument names of the head constant. The example below assumes f is a function with 2 implicit dependent arguments named x and y.

V7	V8
f 1!t1 2!t2 t3	f (x:=t1) (y:=t2) t3
!f t1 t2	@f t1 t2

2.2.4 Inferred subterms

Subterms that can be automatically inferred by the type-checker is now written _

V7	V8
?	_

2.2.5 Universal quantification

The universal quantification and dependent product types are now materialized with the **forall** keyword before the binders and a comma after the binders.

The syntax of binders also changed significantly. A binder can simply be a name when its type can be inferred. In other cases, the name and the type of the variable are put between parentheses. When several consecutive variables have the same type, they can be grouped. Finally, if all variables have the same type parentheses can be omitted.

V7	V8
(x:A)B	forall (x: A), B or forall x: A, B
(x,y:nat)P	forall (x y : nat), P or forall x y : nat, P
(x,y:nat;z:A)P	forall (x y : nat) (z:A), P
(x,y,z,t:?)P	forall x y z t, P
(x,y:nat;z:?)P	forall (x y : nat) z, P

2.2.6 Abstraction

The notation for λ-abstraction follows that of universal quantification. The binders are surrounded by keyword **fun** and ⇒ (=> in ascii).

V7	V8
`[x,y:nat; z](f a b c)`	`fun (x y:nat) z => f a b c`

2.2.7 Pattern-matching

Beside the usage of the keyword pair **match**/**with** instead of **Cases**/**of**, the main change is the notation for the type of branches and return type. It is no longer written between < > before the **Cases** keyword, but interleaved with the destructured objects.

The idea is that for each destructured object, one may specify a variable name (after the **as** keyword) to tell how the branches types depend on this destructured objects (case of a dependent elimination), and also how they depend on the value of the arguments of the inductive type of the destructured objects (after the **in** keyword). The type of branches is then given after the keyword **return**, unless it can be inferred.

Moreover, when the destructured object is a variable, one may use this variable in the return type.

V7	V8

`Cases n of`
`  O => O`
`| (S k) => (1) end`
`Cases m n of`
`  0 0 => t`
`| ...  end`
`<[n:nat](P n)>Cases T of ...  end`
`<[n:nat][p:(even n)]~(odd n)>Cases p of`
`  ...`
`end`
	`match n with`
	`  0 => 0`
	`| S k => 1 end`
	`match m, n with`
	`  0, 0 => t`
	`| ...  end`
	`match T as n return P n with ...  end`
	`match p in even n return ~ odd n with`
	`  ...`
	`end`

The annotations of the special pattern-matching operators (**if**/**then**/**else**) and **let()** also changed. The only restriction is that the destructuring **let** does not allow dependent case analysis.

V7	V8
`<[n:nat;x:(I n)](P n x)>if t then t1` `else t2`	`if t as x in I n return P n x then t1` `else t2`
`<[n:nat](P n)>let (p,q) = t1 in t2`	`let (p,q) in I n return P n := t1 in t2`

2.2.8 Fixpoints and cofixpoints

An easier syntax for non-mutual fixpoints is provided, making it very close to the usual notation for non-recursive functions. The decreasing argument is now indicated by an annotation between curly braces, regardless of the binders grouping. The annotation can be omitted if the binders introduce only one variable. The type of the result can be omitted if inferable.

V7	V8
`Fix plus{plus [n:nat] : nat -> nat := [m]...}`	`fix plus (n m:nat) {struct n}: nat := ...`
`Fix fact{fact [n:nat]: nat := Cases n of O => (1) | (S k) => (mult n (fact k)) end}`	`fix fact (n:nat) := match n with 0 => 1 | (S k) => n * fact k end`

There is a syntactic sugar for single fixpoints (defining one variable) associated to a local definition:

V7	V8
`let f := Fix f {f [x:A] : T := M} in (g (f y))`	`let fix f (x:A) : T := M in g (f x)`

The same applies to cofixpoints, annotations are not allowed in that case.

2.2.9 Notation for type cast

V7	V8
`O :: nat`	`0 : nat`

2.3 Main changes in tactics w.r.t. V7

The main change is that all tactic names are lowercase. This also holds for Ltac keywords.

2.3.1 Renaming of induction tactics

V7	V8
`NewDestruct`	`destruct`
`NewInduction`	`induction`
`Induction`	`simple induction`
`Destruct`	`simple destruct`

2.3.2 Ltac

Definitions of macros are introduced by **Ltac** instead of **Tactic Definition**, **Meta Definition** or **Recursive Definition**. They are considered recursive by default.

V7	V8
`Meta Definition my_tac t1 t2 := t1; t2.`	`Ltac my_tac t1 t2 := t1; t2.`

Rules of a match command are not between square brackets anymore.

Context (understand a term with a placeholder) instantiation **inst** became **context**. Syntax is unified with subterm matching.

V7	V8
`Match t With [C[x=y]] -> Inst C[y=x]`	`match t with context C[x=y] => context C[y=x] end`

Arguments of macros use the term syntax. If a general Ltac expression is to be passed, it must be prefixed with "`ltac :`". In other cases, when a ´ was necessary, it is replaced by "`constr :`"

V7	V8
`my_tac '(S x)`	`my_tac (S x)`
`my_tac (Let x=tac In x)`	`my_tac ltac:(let x:=tac in x)`
`Let x = '[x](S (S x)) In Apply x`	`let x := constr:(fun x => S (S x)) in apply x`

`Match Context With` is now called `match goal with`. Its argument is an Ltac expression by default.

2.3.3 Named arguments of theorems (*bindings*)

V7	V8
`Apply thm with x:=t 1:=u`	`apply thm with (x:=t) (1:=u)`

2.3.4 Occurrences

To avoid ambiguity between a numeric literal and the optionnal occurence numbers of this term, the occurence numbers are put after the term itself and after keyword **as**.

V7	V8
`Pattern 1 2 (f x) 3 4 d y z`	`pattern f x at 1 2, d at 3 4, y, z`

2.3.5 `LetTac` and `Pose`

Tactic `LetTac` was renamed into `set`, and tactic `Pose` was a particular case of `LetTac` where the abbreviation is folded in the conclusion[2].

V7	V8
`LetTac x = t in H`	`set (x := t) in H`
`Pose x := t`	`set (x := t)`

`LetTac` could be followed by a specification (called a clause) of the places where the abbreviation had to be folded (hypothese and/or conclusion). Clauses are the syntactic notion to denote in which parts of a goal a given transformation shold occur. Its basic notation is either ***** (meaning everywhere), or *hyps* `|-` *concl* where *hyps* is either ***** (to denote all the hypotheses), or a comma-separated list of either hypothesis name, or (`value of` H) or (`type of` H). Moreover, occurrences can be specified after every hypothesis after the **at** keyword. *concl* is either empty or *****, and can be followed by occurences.

V7	V8
`in Goal`	`in |- *`
`in H H1`	`in H1, H2 |-`
`in H H1 ...`	`in * |-`
`in H H1 Goal`	`in H1, H2 |- *`
`in H H1 H2 ... Goal`	`in *`
`in 1 2 H 3 4 H0 1 3 Goal`	`in H at 1 2, H0 at 3 4 |- * at 1 3`

[2]There is a tactic called `pose` in V8, but its behaviour is not to fold the abbreviation at all.

2.4 Main changes in vernacular commands w.r.t. V7

2.4.1 Require

The default behaviour of `Require` is not to open the loaded module.

V7	V8
`Require Arith`	`Require Import Arith`

2.4.2 Binders

The binders of vernacular commands changed in the same way as those of fixpoints. This also holds for parameters of inductive definitions.

V7	V8
`Definition x [a:A] : T := M`	`Definition x (a:A) : T := M`
`Inductive and [A,B:Prop]:  Prop :=`	`Inductive and (A B:Prop):  Prop :=`
`  conj :  A->B->(and A B)`	`  conj :  A -> B -> and A B`

2.4.3 Hints

Both `Hints` and `Hint` commands are beginning with `Hint`.

Command `HintDestruct` has disappeared.

The syntax of *Extern* hints changed: the pattern and the tactic to be applied are separated by a `=>`.

V7	V8
`Hint name := Resolve (f ?  x)`	`Hint Resolve (f _ x)`
`Hint name := Extern 4 (toto ?)  Apply lemma`	`Hint Extern 4 (toto _) => apply lemma`
`Hints Resolve x y z`	`Hint Resolve x y z`
`Hints Resolve f :  db1 db2`	`Hint Resolve f :  db1 db2`
`Hints Immediate x y z`	`Hint Immediate x y z`
`Hints Unfold x y z`	`Hint Unfold x y z`

2.4.4 Implicit arguments

`Set Implicit Arguments` changed its meaning in V8: the default is to turn implicit only the arguments that are *strictly* implicit (or rigid), i.e. that remains inferable whatever the other arguments are. For instance `x` inferable from `P x` is not strictly inferable since it can disappears if `P` is instanciated by a term which erases `x`.

V7	V8
`Set Implicit Arguments`	`Set Implicit Arguments.` `Unset Strict Implicits.`

However, you may wish to adopt the new semantics of `Set Implicit Arguments` (for instance because you think that the choice of arguments it sets implicit is more “natural” for you).

2.5 Changes in standard library

Many lemmas had their named changed to improve uniformity. The user generally do not have to care since the translators performs the renaming.

Type `entier` from fast_integer.v is renamed into `N` by the translator. As a consequence, user-defined objects of same name `N` are systematically qualified even tough it may not be necessary. The following table lists the main names with which the same problem arises:

V7	V8
`IF`	`IF_then_else`
`ZERO`	`Z0`
`POS`	`Zpos`
`NEG`	`Zneg`
`SUPERIEUR`	`Gt`
`EGAL`	`Eq`
`INFERIEUR`	`Lt`
`add`	`Pplus`
`true_sub`	`Pminus`
`entier`	`N`
`Un_suivi_de`	`Ndouble_plus_one`
`Zero_suivi_de`	`Ndouble`
`Nul`	`N0`
`Pos`	`Npos`

2.5.1 Implicit arguments

Main definitions of standard library have now implicit arguments. These arguments are dropped in the translated files. This can exceptionally be a source of incompatibilities which has to be solved by hand (it typically happens for polymorphic functions applied to `nil` or `None`).

2.5.2 Logic about `Type`

Many notations that applied to `Set` have been extended to `Type`, so several definitions in `Type` are superseded by them.

V7	V8
`x==y`	`x=y`
`(EXT x:Prop | Q)`	`exists x:Prop, Q`
`identityT`	`identity`

3 A guide to translation

Here is a short description of the tools involved in the translation process:

`coqc -translate` is the automatic translator. It is a parser/pretty-printer. This means that the translation is made by parsing every command using a parser of old syntax, which is printed using the new syntax. Many efforts were made to preserve as much as possible of the quality of the presentation: it avoids expansion of syntax extensions, comments are not discarded and placed at the same place.

`translate-v8` (in the translation package) is a small shell-script that will help translate developments that compile with a Makefile with minimum requirements.

3.1 Preparation to translation

This step is very important because most of work shall be done before translation. If a problem occurs during translation, it often means that you will have to modify the original source and restart the translation process. This also means that it is recommended not to edit the output of the translator since it would be overwritten if the translation has to be restarted.

3.1.1 Compilation with `coqc -v7`

First of all, it is mandatory that files compile with the current version of Coq (8.0) with option `-v7`. Translation is a complicated task that involves the full compilation of the development. If your development was compiled with older versions, first upgrade to Coq V8.0 with option `-v7`. If you use a Makefile similar to those produced by `coq_makefile`, you probably just have to do

`make OPT="-opt -v7"` or `make OPT="-byte -v7"`

When the development compiles successfully, there are several changes that might be necessary for the translation. Essentially, this is about syntax extensions (see section below dedicated to porting syntax extensions). If you do not use such features, then you are ready to try and make the translation.

3.2 Translation

3.2.1 The general case

The preferred way is to use script `translate-v8` if your development is compiled by a Makfile with the following constraints:

- compilation is achieved by invoking make without specifying a target
- options are passed to Coq with make variable COQFLAGS that includes variables OPT, COQLIBS, OTHERFLAGS and COQ_XML.

These constraints are met by the makefiles produced by `coq_makefile`

Otherwise, modify your build program so as to pass option `-translate` to program `coqc`. The effect of this option is to ouptut the translated source of any `.v` file in a file with extension `.v8` located in the same directory than the original file.

3.2.2 What may happen during the translation

This section describes events that may happen during the translation and measures to adopt.

These are the warnings that may arise during the translation, but they generally do not require any modification for the user: Warnings:

- `Unable to detect if` *id* `denotes a local definition`
  This is due to a semantic change in clauses. In a command such as `simpl in H`, the old semantics were to perform simplification in the type of `H`, or in its body if it is defined. With the new semantics, it is performed both in the type and the body (if any). It might lead to incompatibilities

- `Forgetting obsolete module`
  Some modules have disappeared in V8.0 (new syntax). The user does not need to worry about it, since the translator deals with it.

- `Replacing obsolete module`
  Same as before but with the module that were renamed. Here again, the translator deals with it.

3.3 Verification of the translation

The shell-script `translate-v8` also renames `.v8` files into `.v` files (older `.v` files are put in a subdirectory called `v7`) and tries to recompile them. To do so it invokes `make` without option (which should cause the compilation using `coqc` without particular option).

If compilation fails at this stage, you should refrain from repairing errors manually on the new syntax, but rather modify the old syntax script and restart the translation. We insist on that because the problem encountered can show up in many instances (especially if the problem comes from a syntactic extension), and fixing the original sources (for instance the `V8only` parts of notations) once will solve all occurrences of the problem.

3.4 Particular cases

3.4.1 Lexical conventions

The definition of identifiers changed. Most of those changes are handled by the translator. They include:

- _ is not an identifier anymore: it is tranlated to `x_`
- avoid clash with new keywords by adding a trailing _

If the choices made by translation is not satisfactory or in the following cases:

- use of latin letters
- use of iso-latin characters in notations

the user should change his development prior to translation.

3.4.2 `Case` and `Match`

These very low-level case analysis are no longer supported. The translator tries hard to translate them into a user-friendly one, but it might lack type information to do so[3]. If this happens, it is preferable to transform it manually before translation.

3.4.3 Syntax extensions with `Grammar` and `Syntax`

`Grammar` and `Syntax` are no longer supported. They should be replaced by an equivalent `Notation` command and be processed as decribed above. Before attempting translation, users should verify that compilation with option `-v7` succeeds.

In the cases where `Grammar` and `Syntax` cannot be emulated by `Notation`, users have to change manually they development as they wish to avoid the use of `Grammar`. If this is not done, the translator will simply expand the notations and the output of the translator will use the regular Coq syntax.

[3]The translator tries to typecheck terms before printing them, but it is not always possible to determine the context in which terms appearing in tactics live.

3.4.4 Syntax extensions with `Notation` and `Infix`

These commands do not necessarily need to be changed.

Some work will have to be done manually if the notation conflicts with the new syntax (for instance, using keywords like `fun` or `exists`, overloading of symbols of the old syntax, etc.) or if the precedences are not right.

Precedence levels are now from 0 to 200. In V8, the precedence and associativity of an operator cannot be redefined. Typical level are (refer to the chapter on notations in the Reference Manual for the full list):

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

By default, the translator keeps the associativity given in V7 while the levels are mapped according to the following table:

V7 level	mapped to	associativity
0	0	no
1	20	left
2	30	right
3	40	left
4	50	left
5	70	no
6	80	right
7	85	right
8	90	right
9	95	no
10	100	left

If this is OK, just simply apply the translator.

Associativity conflict Since the associativity of the levels obtained by translating a V7 level (as shown on table above) cannot be changed, you have to choose another level with a compatible associativity.

You can choose any level between 0 and 200, knowing that the standard operators are already set at the levels shown on the list above.

Assume you have a notation

```
Infix NONA 2 "=_S" my_setoid_eq.
```

By default, the translator moves it to level 30 which is right associative, hence a conflict with the expected no associativity.

To solve the problem, just add the "V8only" modifier to reset the level and enforce the associativity as follows:

```
Infix NONA 2 "=_S" my_setoid_eq V8only (at level 70, no associativity).
```

The translator now knows that it has to translate "=_S" at level 70 with no associativity.

Remark: 70 is the "natural" level for relations, hence the choice of 70 here, but any other level accepting a no-associativity would have been OK.

Second example: assume you have a notation

```
Infix RIGHTA 1 "o" my_comp.
```

By default, the translator moves it to level 20 which is left associative, hence a conflict with the expected right associativity.

To solve the problem, just add the "V8only" modifier to reset the level and enforce the associativity as follows:

```
Infix RIGHTA 1 "o" my_comp V8only (at level 20, right associativity).
```

The translator now knows that it has to translate "o" at level 20 which has the correct "right associativity".

Remark: we assumed here that the user wants a strong precedence for composition, in such a way, say, that "f o g + h" is parsed as "(f o g) + h". To get "o" binding less than the arithmetical operators, an appropriated level would have been close of 70, and below, e.g. 65.

Conflict: notation hides another notation Remark: use `Print Grammar constr` in V8 to diagnose the overlap and see the section on factorization in the chapter on notations of the Reference Manual for hints on how to factorize.

Example:

```
Notation "{ x }" := (my_embedding x) (at level 1).
```

overlaps in V8 with notation `{ x : A & P }` at level 0 and with x at level 99. The conflicts can be solved by left-factorizing the notation as follows:

```
Notation "{ x }" := (my_embedding x) (at level 1)
  V8only (at level 0, x at level 99).
```

Conflict: a notation conflicts with the V8 grammar Again, use the `V8only` modifier to tell the translator to automatically take in charge the new syntax.

Example:

```
Infix 3 "@" app.
```

Since `@` is used in the new syntax for deactivating the implicit arguments, another symbol has to be used, e.g. `@@`. This is done via the `V8only` option as follows:

```
Infix 3 "@" app V8only "@@" (at level 40, left associativity).
```

or, alternatively by

```
Notation "x @ y" := (app x y) (at level 3, left associativity)
  V8only "x @@ y" (at level 40, left associativity).
```

Conflict: my notation is already defined at another level (or with another associativity) In V8, the level and associativity of a given notation can no longer be changed. Then, either you adopt the standard reserved levels and associativity for this notation (as given on the list above) or you change your notation.

- To change the notation, follow the directions in the previous paragraph
- To adopt the standard level, just use `V8only` without any argument.

Example:

```
Infix 6 "*" my_mult.
```

is not accepted as such in V8. Write

```
Infix 6 "*" my_mult V8only.
```

to tell the translator to use * at the reserved level (i.e. 40 with left associativity). Even better, use interpretation scopes (look at the Reference Manual).

3.4.5 Strict implicit arguments

In the case you want to adopt the new semantics of `Set Implicit Arguments` (only setting rigid arguments as implicit), add the option `-strict-implicit` to the translator.

Warning: changing the number of implicit arguments can break the notations. Then use the `V8only` modifier of `Notation`.