Coq quick reference

Meta variables	Usage	Meta variables	Usage
c	constructor	*P, Q*	terms used as propositions
db	identifier for a hint database	*s*	string
G, H	hypotheses	*scope*	identifier for a notation scope
M,	module identifiers	*t, u*	arbitrary terms
m	module values	*tactic*	Ltac terms with tactic values
n	numbers	*v, w, x, y*	identifiers
p	patterns	*V, W, X, Y*	terms used as types

Category	Example	Description
Sorts	Prop	Universe of logical propositions.
	Set	Universe of program types or specifications.
	Type	Type of Set and Prop.
Types	forall *x* : *X*, *Y* : *Univ*	Dependent product type (*Y* can refer to *x*).
	X -> *Y* : *Univ*	Arrow type (non-dependent product), same as forall _: *X*, *Y*.
Declarations and definitions	CoFixpoint *x* (*y* : *Y*) : *X* := *t*.	As for Fixpoint but defines co-recursive functions.
	CoInductive *X* : *Univ* := | *v* : *V*	As for Inductive but defines co-inductive type.
	Declare Module *M* : *V*.	Declare a module of a particular module type.
	Definition *x* : *X* := *t*.	Constant definition.
	Definition *x* (*v* : *V*) : *X* := *t*.	Non-recursive function definition. Surround arg with braces instead of parens to make it implicit. Use @*x* to get version of function *x* with all arguments made explicit.
	End *M*.	Finish definition of module or module type *M*.
	Fixpoint *x* (*v* : V) : *X* := *t*.	As for Definition but *t* can make recursive calls to *x*.
	Fixpoint *x* (*v* : *V*) : *X* := *t* with *y* (*w* : *W*) : *Y* := *u*.	Mutually recursive function definitions.
	Import *M*.	Import the names of module *M* so they can be used without qualification.

Category	Example	Description
	Inductive *X* : *Univ* := | *v* : *V*	Inductive type with instances defined by constructors, including *y* of type *Y*.
	Inductive *X* : *Univ* := | *v* : *V* ... with *Y* : Univ := | *w* : *W*	Mutually recursive inductive types *X* and *Y*.
	Let *x* : *X* := *t*.	Local definition.
	Module *M* (*x* : *X*).	Start definition of parameterised module *M*.
	Module Type *V*.	Start definition of module type *V*.
	Module *M* := *m*.	Define *M* to be the module *m*
	Parameter *x* : *X*.	Global constant declaration (also Axiom).
	Record *x* (*v* : *V*) : *Univ* := *y* { *w* : *W*; ... }.	Define a dependent record type, i.e., an inductive type *x* with a single constructor *y* which has arguments *w*. The *w* are also projection functions for extracting argument values from a value of this type.
	Require *M*.	Load the file containing module *M*.
	Require Import *M*.	As for Require and import names of *M*.
	Require Export *M*.	As for Require Import and indicate that *M* should be imported if the current module is imported.
	Variable *x* : *X*.	Local variable declaration (also Hypothesis).
	Variables *x* : *X* *y* : *Y*.	Local variable declarations (also Hypotheses).
Terms	fun *x* : *X* => *t*	Non-recursive function.
	fix *x* (*v* : *V*) : *X* := *t*	Recursive function.
	t t_1 ... t_n	Function call.
	_	Anonymous variable in pattern or ask for argument to be inferred in function call.
	let *x* := *t* in *u*	Local binding.
	match *t* *return* with | p_1 => t_1 | ... end	Dependent pattern matching term selection where *return* expresses the type of the match expression. *return* can be omitted if it is the common type of all of the branches.
	return *T*	Return type is *T*.

Category	Example	Description
	as *x*	Prepend to return type so that *T* can refer to the matched term *t* by name *x*.
	in *U* u_1 u_2 ...	Prepend to return type so that *T* can refer to the type *U* u_1 u_2 ... of the matched term *t*.
	as *x* in *U* u_1 u_2 ...	Combination of above.
	if *t* then *u* else *v*	If *t* is of type Bool, same as match *t* with | true => *u* | false => *v* end. Works for all types with exactly two constructors.
	admit	Term that fits in any hole.
Evaluation	Eval *tactic* in *t*.	Normalisation of *t* using conversion *tactic* chosen from red, hnf, compute, simpl, cbv, lazy, unfold, fold, pattern.
Theorems	Theorem *x* : *X*.	Theorem definition (also Example, Lemma).
Proofs	Proof.	Start proof of theorem (optional).
	Admitted.	Admission (stands for proof).
	Defined.	As for Qed but mark definition as transparent so it can be unfolded.
	Guarded.	Report if it is possible to finish a proof yielding a properly guarded co-inductive proof term.
	Qed.	Build, check and save opaque proof term.
Proof steps	Show.	Print current goals.
	Show *n*.	Print *nth* goal.
	Show Proof.	Print the current proof term.
	tactic.	Apply tactic to first goal.
	n:tactic.	Apply tactic to *nth* goal.
	tactic in *H.*	Apply tactic to hypothesis *H.*
	tactic in **.*	Apply tactic to first goal and all hypotheses.
	Undo [*n*].	Undo *n* single steps (default: 1).
	Restart.	Go back to beginning of proof attempt.
	Abort.	Abort proof attempt.
Printing	Check *t*.	Check if well-formed, print type.
	Print *x*.	Print value and type.
	Print Assumptions *x*.	Print the axioms, parameters and variables that the theorem or definition *x* depends on.

Category	Example	Description
Searching	SearchAbout *t*.	List results whose conclusion is (*t* ...) where *t* is usually just an identifier.
	SearchPattern (*p*).	List theorems whose conclusion or final hypthesis matches *p* (e.g, _ + _ + <= _).
	SearchRewrite (*p*).	List theorems whose conclusion is an equality where one side matches *p.*
	?*id*	Meta-variable in non-linear pattern.
	_	Wildcard pattern.
Hints	Hint *hintdef* : *db ...*	Declare the specified hints in the given databases (default: core).
	Hint Local *hintdef* : *db ...*	Declare a local hint that is not to be exported.
	Constructors *id*	Add all of the constructors of type *id.*
	Extern *n* [*pattern*] => *tactic*	Add *tactic* as a hint for when *pattern* matches (with cost *n).*
	Immediate *t*	Add apply *t*; trivial (cost 1).
	Opaque *x*	Add an opacity hint about *x.*
	Resolve *t*	Add apply *t* (cost number of subgoals generated).
	Rewrite t_1 t_2 ...	Add equalities t_i to rewrite database. Also Rewrite <- t_1 t_2 ... to add rewrite from right to left.
	Transparent *x*	Add a transparency hint about *x.*
	Unfold *x*	Add unfold *x* (cost 4).
Notation	Notation "..." := *t* (at level *n*, left/ right associativity) : *scope*	Define new notation in given scope, syntax from string pattern, defined to be term *t* with priority *n* (from 0 to 100) and given associativity.
	Notation "..." := *t*.	Define new notation in current scope with default priority and associativity.
	Reserved Notation "..." (at level *n*, left/right associativity) : *scope*	Reserve notation without giving its definition so it can be used to define its own meaning.
	Inductive ... where "..." := *t* : scope	Provide a declaration of a reserved notation while using it in an inductive definition
Scopes	Close Scope *scope.*	Remove all occurrences of *scope* from the notation interpretation stack. Same modifiers as Open Scope.
	Locate "*".	Locate interpretations of notation "*".

Category	Example	Description
	Open Scope *scope*.	Add *scope* to the stack of notation interpretation scopes. As shown, the change does not survive the end of a section where it occurs. Add Global modifier to survive sections or Local modifier to avoid exporting from module.
	Print Scope *scope*.	Print notations from *scope.*
	t %*scope*	Term with notation interpreted in *scope.*
Environments and Sections	Reset Initial.	Reset to initial environment and empty context.
	Reset *x*.	Reset everything since and including *x.*
	Section *x*.	Begin nested section called *x*.
	End *x*.	End nested section *x* and add variables and hypotheses from the section as extra arguments to definitions from the section.
Extraction	Extraction *x*.	Extract a single constant or module.
	Recursive Extraction *x y ...* .	Recursively extract *ids* and their dependencies.
	Extraction "*file*" *x y ...* .	Recursively extraction to a file.
	Extraction Library *x*.	Extract file *x*.v to *x*.ml.
	Recursive Extraction Library *x*.	Recursive extraction of library *x* and dependencies.
	Extraction Language *language*.	Set language for extraction to Ocaml, Haskell or Scheme.
Miscellaneous	Implicit Arguments *x* [*v w ...*].	Make arguments *v w ...* of *x* implicit.
	x ... (v := t) ...	Call *x* giving an explicit value for the implicit argument *v*.
Options	Set/Unset *option*.	Toggle an option.
	Printing All	Whether to show full details of terms.
	Printing Notations	Whether to print using notations or not.
	Implicit Arguments	Auto infer arguments at function definition.

Library

Type	Notation	Constructors
and (P Q: Prop) : Prop	P ∧ Q	conj : P -> Q -> P ∧ Q
bool : Set		true : bool false : bool
eq (X : Type) (x : X) : X -> Prop	x = y	eq_refl : x = x
ex (X : Set) (P : X -> Prop)	exists x : X, P	ex_intro : forall x : X, P x -> exists x, P x
False : Prop		
nat : Set	Numerics in nat_scope	O : nat S : nat -> nat
or (P Q: Prop)	P ∨ Q	or_introl : P -> P ∨ Q or_intror : Q -> P ∨ Q
sig (X : Type) (P : X -> Prop) : Type	{ x : X | P }	exist : forall x : X, P x -> sig P
sumbool (P : Prop) (Q : Prop) : Set	{ P } + { Q }	left : P -> { P } + { Q } right : Q -> { P } + { Q }
sumor (X : Type) (P : Prop) : Type	X + { P }	inleft : X -> X + { P } inright : P -> X + { P }
unit : Set		tt : Set
True : Prop		I : True

Definition Type	Description
all : forall X : Type, (X -> Prop) -> Prop	Universal quantification
iff : Prop -> Prop -> Prop	If and only if
not : Prop -> Prop	Logical negation
proj1 : forall P Q : Prop, P ∧ Q -> P	Left projection
proj1_sig : forall (X : Type) (P : X -> Prop), {x | P x} -> X	Left projection from dependent sum
proj2 : forall P Q : Prop, P ∧ Q -> Q	Right projection
proj2_sig : forall (X : Type) (P : X -> Prop) (e : {x | P x}), P (proj1_sig e)	Right projection from dependent sum

Tactics

Tactics work on the conclusion of the current goal by default. Use "*tactic* in *H*" to apply to hypothesis *H* or "*tactic* in *" to apply to conclusion and all hypotheses.

Tactics including apply, assumption, auto, constructor, destruct, exact, exists, induction, left, split and right also have existential versions (e.g., eapply, eassumption, etc) which do not fail if an instantiation of a variable cannot be found. Rather, they introduce new existential variables in those positions.

Tactic	Hypotheses	Conclusion	Description
apply *H*	*H* : *u* -> t_2	t_1	Apply deduction backwards, where t_1 unifies with t_2. New conclusion is *u.* Can also use theorems or constructors
apply *H* with *x* := *t ...*			As for apply *H* but with specified bindings in *H*. Can omit "*x* :=" if unambiguous
apply *G* in *H*	*H* : t_1 *G* : t_2 -> *u*		Apply deduction forwards in *H*, where t_1 unifies with t_2 and replace with *u*.
assert *t as H*		*u*	New subgoal *t* called *H* which is to be proven and then used to prove *u.*
assert (*H : t)*			Same as assert *t* as *H.*
assumption	*H* : *t*	*t*	Look for conclusion in hypotheses; if found, same as apply *H.*
auto [*n]*			Solve if possible with combination of intros, apply and reflexivity. Uses current hypotheses plus core hint database. Only go to a depth of *n* (default: 5).
auto using *x, y*, ...			auto with *x, y*, ... added to hint database.
auto with *db ...*			auto using given hint databases as well as default core. *databases* can be * to use all databases
autorewrite with *db ...*			Rewrite according to rewrite databases (built with Hint Rewrite).
case *t*			Simpler than destruct and like elim but uses case-analysis elimination principle not a recursive one.
cbv *flags*			Call by value normalisation where *flags* are from beta, delta, iota or zeta. If no flag is given all are assumed.
change *t*		*u*	Change conclusion to *t* if *t* and *u* are convertible.
clear *H*	H : ...		Remove *H* from context.
cofix *x*			Primitive tactic to start proof by co-induction using hypothesis *x.*
compute			Synonym for cbv beta delta iota zeta.

Tactic	Hypotheses	Conclusion	Description
congruence		*t = u*	Decision procedure for ground equalities and uninterpreted functions.
constructor			Look for constructor *c* that can be applied to solve current goal; if found, same as apply *c.*
contradiction	*H : False*		Look for hypothesis that is equivalent to False; if found, solve goal.
cut *t*		*u*	New goals: *t* -> *u* and *t.*
decide equality		forall *x y* : X, {*x = y*} + {*x* <> y}	Solve a decidable equality.
destruct *t*	*H* : ... *t* ...	... *t* ...	Destruct *t* by cases.
discriminate		*t* <> *u*	Show that *t* and *u* are not equal if they are made using different constructors.
elim *t*			Basic induction that eliminates by choosing appropriate destructor and applying it.
exact *t*		*u*	Prove *u* by giving exact proof term *t.*
exists *t*		*ex X P*	Witness *x* : *X* and *P* : *X* -> Prop, conclusion becomes *P x.*
fail			Always fail to apply
f_equal		f t_1 t_2 = f u_1 u_2	Replace with goals $t_1 = u_1$, $t_2 = u_2$.
field			Extension of ring tactic to rational expressions.
firstorder			First order logic decision procedure.
fix *x n*			Primitive tactic to start proof by induction calling induction hypothesis *x* and acting on premise *n* of current goal.
fold *t*			Replace occurrences of the reduction of *t* (using red).
generalize *t*		... *t* ...	Generalize a goal sub-term *t*, replacing it with a new variable *id* and adding forall *x* to the goal.
generalize dependent *t*			Same as generalize, except also generalizes all hypotheses that depend on *x,* clearing them.
hnf			Replace current conclusion with its head normal form.
idtac [*s* ...]			Identity, print *s* if present.
induction *t*			Induction on *t.*

Tactic	Hypotheses	Conclusion	Description
induction *t* using *x*			Induction on t with non-standard induction principle *x*.
injection *H*	H : t = u		Use injectivity of constructors in *H* to infer equalities between sub-terms of *t* and *u*.
instantiate *n* := *t*	... ?x ...		Refine the *nth* existential variable in the conclusion (numbering from right) with the term *t*.
intro *H*		*t* -> *u*	New hypothesis *H* : *t*, conclusion *u*.
intros [H_1 H_2 ...]		t -> u -> ... -> v	intro H_1, intro H_2, ...
intros [*x y*]		forall x : X, y :Y, t	New hypotheses *x* : *X*, *y* : *Y* conclusion *t*.
intuition			intuition auto with *.
intuition *tactic*			Intuitionistic propositional calculus decision procedure and apply *tactic* to the generated subgoals.
inversion *H*	H : c x = c y		Invert injective constructor *c* to get *x* = *y*.
inversion *H*	H : c x = d y		Invert different constructors to get contradiction.
inversion *H*	H : t		Invert inductively defined *t*, creating one subgoal for each way in which *t* could be constructed (with associated assumptions).
lazy *flags*			Call by need normalisation where *flags* are from beta, delta, iota or zeta. If no flag is given all are assumed
left		E.g., *P* ∨ *Q*	Apply first constructor of a type with two constructors, e.g., apply or_introl.
omega			Decision procedure for first-order logic with Presburger arithmetic (numeric constants, +, -, S, pred, * by constant, =, <>, <=, ∧, ∨, ~ and ->).
pattern *t*			Perform beta expansion on *t* which should be a free sub-term of the current conclusion.
pose *x* := *t*			Add local definition *x* := *t* to hypotheses.
red		forall (*x* : *X*), *y x*	Where *y* is a transparent constant, replace *y* by its definition then beta, iota and zeta reduce.
refine *t*		*u*	Give exact proof term like exact but with some holes marked with "_".
reflexivity		*t* = *u*	Simplify then remove equality.

Tactic	Hypotheses	Conclusion	Description
remember *t* as *x*		*... t ...*	Add hypothesis *x = t* to all subgoals and replace *t* in goal with *x.*
rename *H* into *G*	*H: ...*		Rename hypothesis *H* to *G.*
replace *t* with *u*		*... u ...*	Replace all occurrences of *t* by *u* and add new subgoal *t = u.*
rewrite *H*	*H : t = u*	*... t ... t ...*	Same as rewrite -> *H*
rewrite -> *H*	*H : t = u*	*... t ... u ...*	Rewrite *t* to *u. H* can also be a previously proven theorem.
rewrite <- *H*	*H : t = u*	*... u ... u ...*	Rewrite *u* to *t.*
right		E.g., *P* ∨ *Q*	Apply second constructor of a type with two constructors, e.g., apply or_intror.
ring			Normalise polynomials over any ring or semi-ring structure.
set *x* := *t*	H : ... t ...	*... t ...*	Replace *t* with *x* in the current goal and add local definition *x* := *t* to the context.
simpl			Simplify by applying beta and iota reduction and expanding transparent constants.
specialize (*H t u ...*)	H : *P* -> ... -> Q		Specialise *H* by applying it to terms *t u ...*
split			intros; apply conj.
subst *x*	*x = t* or *t* = x	*... x ...*	Replace *x* with *t*, clear assumption.
subst			Perform substitution of all equality assumptions.
symmetry		*t = u*	Switch sides to get *u = t.*
tauto			intuition fail.
trivial			Automatically apply basic tactics using only hypotheses with a single premise.
unfold *x, y, ...*			Delta reduce by *x, y, ...* and simplify.

Tacticals

tactic₁; tactic₂	Apply *tactic₁* then apply *tactic₂* to all goals that result.
tactic₁; [*tactic₂*	*tactic₃*	...]	Apply *tactic₁* then apply *tactic₂* to the first resulting goal, *tactic₃* to second goal and so on.
tactic₁ || *tactic₂*	Apply *tactic₁* and if it fails apply *tactic₂* to the original goal.
abstract *tactic*	Similar to solve but also saves auxiliary lemma.
do *n tactic*	Apply *tactic* to the goal *n* times.
first [*tactic₁*	*tactic₂*	...]	Apply the first tactic in the sequence that doesn't fail.
info *tactic*	Behave as *tactic* but display operations for complex tactics such as auto.
progress *tactic*	Fail if *tactic* succeeds without making progress.
repeat *tactic*	Repeatedly apply *tactic* until no goals left or it fails on all subgoals.
solve *tactic*	If *tactic* solves the goal, behave just like *tactic*, otherwise do nothing.
solve [*tactic₁*	*tactic₂*	...]	As for solve but try to solve with the listed tactics in order.
timeout *n tactic*	Stop *tactic* if it hasn't finished after *n* seconds.
try *tactic*	Same as *tactic* || idtac.
(...)	Grouping.

Ltac

Ltac *x v w ...* := *t.* defines a new tactic *x* with arguments *v w ...* defined by the Ltac term *t.*

Terms	constr:(*t*)	Construct the Gallina term *t.*
	fresh *s*	Evaluate to an identifier unbound in the goal with the new name being *s* with a number appended to make it unique if necessary.
	x	Value of the bound variable *x*.
	?*x*	Value of variable ?*x* previously bound in a pattern.
	let [rec] *x v w ....* := *t* in *u*	Bind *x* with arguments *v w ...* to *t* in *u.* If rec is present *t* can contain recursive references to *x*.
	match goal with | *p* => *tactic* | ... *end.*	Match the current proof state against the patterns in the order given and if a match is found, apply the corresponding tactic. If the tactic fails, backtrack to find other matches for the current pattern, if any. If there are no (more) matches to a pattern, go to the next clause. Hypotheses are matched in newest to oldest order.
	match reverse goal with ... end.	As for match goal with but match hypotheses from oldest to newest.
	match *t* with | *p* => *tactic* | ... end.	Non-linear pattern matching of an Ltac term.
	type of *t*	The type of Gallina term *t.*
Patterns	|- *p*	Match against top of the current conclusion.
	|- context [*p*]	Match against any sub-term of the current conclusion.
	p_1 |- p_2	Match p_1 against the hypotheses and p_2 against the conclusion.
	t	Term-valued pattern.
	?*x*	Variable pattern.
	$H_1 : p_1, H_2 : p_{2, ...}$	Match patterns from right to left against hypotheses.