Coq Cheat Sheet for CS3234 (Aquinas Hobor and Martin Henz)

Note. This is not meant to be comprehensive or stand on its own, but just to give you some idea for what is going on in an informal manner, as well as give you a cheat sheet when you want to remind yourself what, say, "rewrite" does.

Reserved Symbols

- **.**
  All commands in Coq end with a "."
- :
  Used to connect a *value/variable* in Coq with a *type*. (t : Term) means that "t" has the type "Term". See "Type", below. Often ":" can be left out and Coq will guess it correctly. It is useful when debugging to add types to various places using the ":" symbol (experiment!).
- **:=**
  Used when making definitions to separate the name of the definition from its implementation.
- **=>**
  Used when defining functions or when using match/case analysis.
- **,**
  Used to separate quantifiers like "forall" and "exists" from their bodies. Used in pairs, etc.
- **|**
  Used to separate case analysis in "match ... with ... end"

Commands

- **Section**
  Signals the beginning of a named section. Use "End" to close the section when done.
- **End**
  Closes a named section (and other things like modules).
- **Parameter**
  Asserts (axiomatically) the existence of a named object with specified type. For example, "Parameter foo : Type -> Type -> Prop -> Type" says that there is something called foo that has the type "Type -> Type -> Prop -> Type". See "Type" and "->" below.
- **Definition**
  Creates a new definition. "Definition bar : ty := def" creates a new definition "bar", which must have type "ty", and which has implementation "def". Most of the time you can leave out the ": ty" part and Coq will guess. If you want a definition to be parameterized, then you can write, e.g., "Definition addN (n : nat) : nat -> nat := fun m => n + m". Now we have defined "addN(x)", which itself is a function from naturals to naturals. "addN 3" is the function that adds 3 to its argument "(addN 3) 7 = 10"; "addN 17" is the function that adds 17 to its argument.
- **Notation**
  Introduces a "shorthand" or "pretty" way of saying things. Exact construction is a bit tricky and if you are curious then experiment starting with the ones we have given you first.

- **Check**
  Causes Coq to print out the type of a given expression. For example, “Check Term” will return “Type” and “Check non” will return “Term -> Term”. See “Type” below.
- **Print**
  Causes Coq to print out a given definition. For example, “Print AllGreeksHumansConverted” will cause Coq to print “AllGreeksHumansConverted : CategoricalProposition := convert (All Greeks are humans)”. Note that Print also gives you the type of the definition.
- **Axiom**
  Asserts the existence of some named object of specified type. Very similar to “Parameter”, above, but usually used to assert the existence of objects of type “Prop”.
- **Lemma**
  Starts proof-editing mode with a given named goal (which usually has type “Prop”).
- **Proof**
  Not strictly required, but good form to put just after the statement of your lemma to separate the lemma from its proof.
- **Qed**
  Required to end a proof once “Proof Completed” appears.
- **Various Tactic Creation Commands (ltac, tactic notation, ...)**
  Experiment if you like, but manipulating the Coq tactic system is beyond the scope of this course. See the Coq documentation for details and/or examine ones we provide.

Built-in types/constructors

- **Type**
  Every object in Coq is associated with some type. The technicalities can get a bit heavy, but to a first approximation, an object’s type is the *set* that the object comes from. For example, there is a type “nat” which contains the natural numbers, and “3 : nat” means that “3” has type “nat”, or roughly that “3” is in the set “nat”. Working on our informal style, “Type” is the type of types! In other words, “nat : Type”. (Things start getting a bit complicated when we start wondering about things like “Type : Type”.)
- **Prop**
  “Prop” is Coq’s built-in type for logical propositions. We will use “Prop” quite a lot, in axioms, lemmas, etc. For example, the expression “forall A, A -> A” has type “Prop”, that is, “(forall A, A -> A) : Prop”. Of course, “Prop : Type”, etc. but now we start getting complicated again. We have already seen lots of ways to build Props:
  - **->**
    Used for (meta-)implication, such as the horizontal lines of proof rules. Relevant tactics include “intro/intros”, “generalize”, “apply”, “spec”, etc.
  - **=**
    Used for equalities. Relevant tactics include “rewrite”, “reflexivity”, etc.
  - **forall**
    Used for universal quantification. “forall” takes an argument, which is the name of some bound variable (which can be restricted to a type with the “:” operator although usually Coq is able to guess the type), followed by a “,”, followed by the

    body of the quantification. Useful tactics include “intro(s)”, “generalize”, “apply”, “spec”, etc.
  - **/**
    Used to specify “and” (i.e., conjunction). Use tactics “split” and “destruct”.
  - **<->**
    Bi-implication. “P <-> Q” is equivalent to “(P -> Q) /\ (Q -> P)”. Same tactics used for “/\” and “->” then apply.

- **Record**
  Used to define a new object, in the case of term logic new types. Can be used in several different ways. For “Quantity” and “Quality”, used to introduce disjoint (tagged) sums – that is, constructors for the types that are not equal (universal is not the same as particular!). For CategoricalProposition, used in a way that is somewhat similar to “struct” in C or “Object” in Java – to group several objects together. In this case one is given an constructor just as with “universal” for “Quantity” – the constructor for “CategoricalProposition” is “cp”, followed by a series of objects of the given type (Quantity, Quality, Term, Term).
- **fun**
  Used to define a function. “fun x : ty => def” defines a (anonymous) function (lambda-term) with argument “x” of type “ty” (which can normally be left out) and body “def”.
- **match ... with ... end**
  Used for case analysis. “match expr with case1 => body1 | ... | caseN => bodyN end” does pattern-matching on the expression “expr” into cases “case1” ... “caseN”; when “casei” holds, then the match evaluates to “bodyi”. Cases are split with “|” and the last case is concluded with “end”.

Built-in logical operators

- **->, =, forall, /\\, <-**
  See “Prop” above.

Built-in Tactics

- **unfold** *d*
  Replace a term in the goal with its definition. To do the replacement in a hypothesis instead of the goal, use “unfold d in H”
- **intro, intros**
  Move a universal quantification or implication from beneath the bar to a hypothesis above the bar. “intro” does this only once while “intros” does it as many times as it can without working “too hard”.
- **spec**
  If we have a hypothesis “H : A -> B” and another hypothesis (or axiom, etc.) “H1 : A”, then “spec H H1” will simplify “H” into “B”. If “H : A -> B -> C -> D”, and “H1 : A”, “H2 : B”, “H3 : C”, then “spec H H1 H2 H3” will simplify “H” into “B”. Experiment if you like. Also works with “forall”s.
- **generalize**
  Similar to “spec” but more powerful. In the same setup as “spec” above, “generalize H H1

H2” in the context of a goal G would transform the goal into “(C -> D) -> G”. One could the use “intro” to bring the new hypothesis “C -> D” above the bar. Experiment with this.

- **apply**
  If we have a hypothesis “H : A -> B” (or axiom, previous lemma), and our goal is “B”, then “apply H” will solve our goal and present us with a new goal, “A”. If we need to provide some additional hints as to how to use “H”, we can use “apply H with (...)”. See the text for some example, and the Coq documentation for more detail. If we want to use “apply” in a goal, use “apply H in H1” or “apply H in H1 with ...”.
- **rewrite**
  Used to substitute an equality into the goal. If we have “H : A = B” as a hypothesis (or axiom, previous lemma, etc.), and our goal contains “... A ...” then “rewrite H” will transform our goal into “... B ...”. If we want to go the other way, we can use “rewrite <- H”, which will transform our goal back to “... A ...”. If we want to rewrite in a hypothesis then we can use “rewrite H in H1” or “rewrite <- H in H1”. Sometimes we need to do this multiple times if there are multiple “A”s in the term. We can also use “rewrite H in *” to rewrite H everywhere.
- **reflexivity**
  Proves the goal “X = X”.
- **trivial**
  Tries a few “really simple” things, like “assumption” and “reflexivity”. Shorter to type than either.
- **assumption**
  If you have a hypothesis equal to the goal then it can prove the goal.
- assert
- **split**
  Used to prove an “and” (conjunction, /\) in the goal.
- **destruct**
  Tactic useful in a large number of ways. Our initial encounter is to “break up” an “and” (conjunction, /\) in a hypothesis.
- **admit**
  Prove anything. Useful for instructors to give problem sets and when you want to prove goals in different orders than Coq gives you before coming back to solve the previous goals. Automatically get zero points for a homework problem when you include an “admit” in a solution.

Custom tactics for term logic: These are defined and explained in the text.

- **eliminateConversion1**
- **eliminateConversion2**
- **eliminateContraposition1**
- **eliminateContraposition2**
- **eliminateObversion**