

## Beginning Coq Reference Sheet

### Administrivia, Syntax

- ◇ Every Coq command ends with a period.
- ◇ The phrase *Theorem T identifying statement S is proven by P* is formalised as
 

```
Theorem T : S. (* T is only a name and can be used later. *)
Proof.
P (* See the current state of the proof in the CoqIde by clicking, in the toolbar,
   on the green arrow pointing at a yellow ball;
   or do "C-c C-Enter" in Proof General with Emacs. *)
Qed.
```
- ◇ Instead of **Theorem**, you may also see proofs that start with **Example**, **Lemma**, **Remark**, **Fact**, **Corollary**, and **Proposition**, which all mean the SAME thing. This difference is mostly a matter of style.
- ◇ The command **Admitted**, in-place of **Qed**, can be used as a placeholder for an incomplete proof or definition.
  - Useful if you have a subgoal that you want to ignore for a while.
- ◇ **Abort**, in-place of **Qed**, is used to give up on a proof for the moment, say for presentation purposes, and it may be begun later with no error about theorems having the same name.

**Comments** (\* I may be a multiline comment. \*)

**Stand alone commands** As top-level items, we may make commands for:

**Normalisation** **Compute X** executes all the function calls in **X** and prints the result.

**Type inspection** Command **Check X.** asks Coq to print the type of expression **X**.

**Introduce local definitions** Two ways,

- ◇ Simple alias: **pose (new\_thing := complicated\_expression).**
- ◇ More involved: Write tactic **assert (x : X).** to define a new identifier **x** for a proof of **X** which then follows, and is conventionally indented.

**Imports** Loading definitions from a library,

**Require Import Bool.**

### Pattern matching with destruct

We case on value **e** by **destruct e as [ a00 ... am0 | ... | a0n ... amn ]**, which gives us **n** new subgoals corresponding to the number of constructors that could have produced **e** such that the *i*-th constructor has arguments **ai0, ..., ak[U+1D62]**.

- ◇ The intros pattern **as [ ... ]** lets us use any friendly names of our choosing. We may not provide it at the cost of Coq's generated names for arguments.
- ◇ Many proofs pattern match on a variable right after introducing it, **intros e. destruct e as [ ... ]**, and this is abbreviated by the intro pattern: **intros [ ... ]**.
- ◇ If there are no arguments to name, in the case of a nullary construction, we can just write **[]**.

### Simple Tactics

**exact** If the subgoal matches an *exact* hypothesis, Then use **exact <hyp\_name>.**

**simpl** If the current subgoal contains a function call with all its arguments, **simpl** will execute the function on the arguments.

- ◇ Sometimes a call to **unfold f**, for a particular function **f**, is needed before **simpl** will work.

**Modus ponens, or function application** If we have **imp : A -> B**, **a : A** then **imp a** is of type **B**. This also works if the **imp** contains **forall**'s.

**Local tactic application** **t in s** performs the tactic **t** only within the hypothesis, term, **s**. For example, **unfold defnName in item** performs a local rewrite.

### intros tactic: '∀, ⇒' introduction

- ◇ To prove a statement of the form (**forall A : Prop, Q**) we use the  $\forall$ -introduction tactic, supplied with a name for the variable introduced, as in **intros A.**
- ◇ To prove an implication **A ⇒ B** we again use, say, **intros pf\_of\_A.**
- ◇ The **intros** command can take any positive number of arguments, each argument stripping a **forall**, (or **->**), off the front of the current subgoal.

### Notation, Definition, and the tactics fold and unfold

**Definition** is a vernacular command that says two expressions are interchangeable. Below (**not A**) and **A -> False** are declared interchangeable.

**Definition not (A:Prop) := A -> False.**

**Notation "~ x" := (not x) : type\_scope.**

Tactics **unfold defnName** and **fold defnName** will interchange them.

**Notation** creates an operator and defines it as an alternate notation for an expression.

( Use **intros** when working with negations since they are implications! )

### Proof Refinement

Suppose our goal is to prove **B** but we have a proof **imp : A → B**, then if we have an **A** function application suffices. However, when we have no **A** lying about and would like to focus on constructing such an **A** we use **refine (imp \_)** which forces us into constructing a subgoal **A**. It is good practice to then indent proof for the new subgoal.

- ◇ If **imp** has more arguments then **refine** would take more underscores corresponding to the arguments we do not have proofs of; we may place the arguments which we do have access to there and then.
- ◇ Needless to say, a **refine** may occur within a **refine**.

## Examples of Common Datatypes

◇ Prop Type

- A Prop either has a proof or it does not have a proof.
- Coq restricts Prop to being either proven or unproven, rather than true or false.

◇ Sums

```
Inductive or (A B:Prop) : Prop :=
| or_introl : A -> A ∨ B
| or_intror : B -> A ∨ B
where "A ∨ B" := (or A B) : type_scope.
```

◇ Products

```
Inductive and (A B:Prop) : Prop :=
conj : A -> B -> A ∧ B
where "A ∧ B" := (and A B) : type_scope.
```

◇ Naturals

```
Inductive nat : Set :=
| 0 : nat (* Capital-letter 0, not the number zero. *)
| S : nat -> nat.
```

◇ Options

```
Inductive option (A : Type) : Type :=
| Some : A -> option A
| None : option A
```

◇ Lists

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A -> list A -> list A.

Infix ":@" := cons (at level 60, right associativity) : list_scope.
```

## True, False, true, false

The vernacular command **Inductive** lets you create a new type.

- ◇ The empty Prop, having no proofs, is **False**.
- ◇ The top Prop, having a single proof named **I**, is **True**.
- ◇ The **bool** type has two values: **true** and **false**.

```
Inductive False : Prop := .
```

```
Inductive True : Prop :=
| I : True.
```

```
Inductive bool : Set :=
| true : bool
| false : bool.
```

In the boolean library there is a function **Is\_true** which converts booleans into their associated Prop counterparts.

## Existence ∃

```
Inductive ex (A:Type) (P:A -> Prop) : Prop :=
ex_intro : forall x:A, P x -> ex (A:=A) P.
```

```
Notation "'exists' x .. y , p" := (ex (fun x => .. (ex (fun y => p)) ..))
(at level 200, x binder, right associativity,
format "'[' 'exists' '/' ' x .. y , '/' ' p ']'")
: type_scope.
```

Note that the constructor takes 3 arguments: The predicate **P**, the witness **x**, and a proof of **P x**.

If we pose a witness beforehand then **refine (ex\_intro \_ witness \_)**, Coq will infer **P** from the current goal and the new subgoal is the proof that the witness satisfies the predicate.

## Equality, rewrite, and reflexivity

Two operators,

- ◇ **x = y** :> **A** says that **x** and **y** are equal and both have type **A**.
- ◇ **x = y** does the same but let's Coq infer the type **A**.

```
Inductive eq (A:Type) (x:A) : A -> Prop :=
eq_refl : x = x :>A
```

```
where "x = y :> A" := (@eq A x y) : type_scope.
```

```
Notation "x = y" := (x = y :>_) : type_scope.
```

Rather than using **destruct**, most proofs using equality use the tactics **rewrite** <orientation>. If **xEy** has type **x = y**, then **rewrite -> xEy** will replace **x** with **y** in the subgoal, while using orientation **<-** rewrites the other-way, replacing **y** with **x**.

- ◇ This can also be used with a previously proved theorem. If the statement of said theorem involves quantified variables, Coq tries to instantiate them by matching with the current goal.
  - ◇ As with destructing, the pattern **intros eq. rewrite -> eq.** is abbreviated by the intro pattern **intros []**. which performs a left-to-right rewrite in the goal.
- Use the **reflexivity** tactic to discharge a goal of type **x = x**.
- ◇ This tactic performs some simplification automatically when checking that two sides are equal; e.g., it tries **simpl** and **unfold**.

## Discrepancy

Coq uses the operator **<>** for inequality, which really means *equality is unprovable or equality implies False*.

```
Notation "x <> y :> T" := (~ x = y :>T) : type_scope.
Notation "x <> y" := (x <> y :>_) : type_scope.
```

Datatype constructors are necessarily disjoint, hence if we ever obtain a proof **pf** of distinct constructors being equal then we may invoke **discriminate pf** to short-circuit the current goal, thereby eliminating a case that could not have happened.