The Coq Proof Assistant Tutorial

Software Verification

Amy Pritchard
University at Buffalo, SUNY
amypritc@buffalo.edu

Last Update: Winter 2019

CoqIDE v8.10.2 (Updated up to Section 9.3)

CoqIDE v8.7.2 (to be removed once fully updated)

Note: Screenshots and tutorial are based on the CoqIDE for MacOS - there may be differences between OS versions.

Contents

Part I
Introduction

1 The COQ Proof Assistant Web Page

`https://coq.inria.fr`

Here you can find all sorts of info about the COQ Proof Assistant, including the following:

- About COQ (`https://coq.inria.fr/about-coq`)
- Getting COQ (`https://coq.inria.fr/download`)
- Documentation, i.e. books, tutorials, etc. (`https://coq.inria.fr/documentation`)
- Reference Manual (`https://coq.inria.fr/distrib/current/refman/`).
- Standard Library (`https://coq.inria.fr/distrib/current/stdlib/`)
- Community for discussion, contribution, events, etc. (`https://coq.inria.fr/community`)
- News about new releases (`https://coq.inria.fr/news/`)

2 Installation

For general installation options and instructions, go to `https://coq.inria.fr/download`.

To get an installer for the latest version of the CoqIDE for Windows or MacOS, go to `https://github.com/coq/coq/releases/latest` and scroll down to the bottom of the page.

To get the installer for Windows or MacOS for CoqIDE v8.10.2 (used for this tutorial), go to `https://github.com/coq/coq/releases/tag/V8.10.2` and scroll down to the bottom of the page.

To install Coq via OPAM on MacOS or Linux, go to `https://coq.inria.fr/opam-using.html` for step by step instructions.

3 The Coq IDE

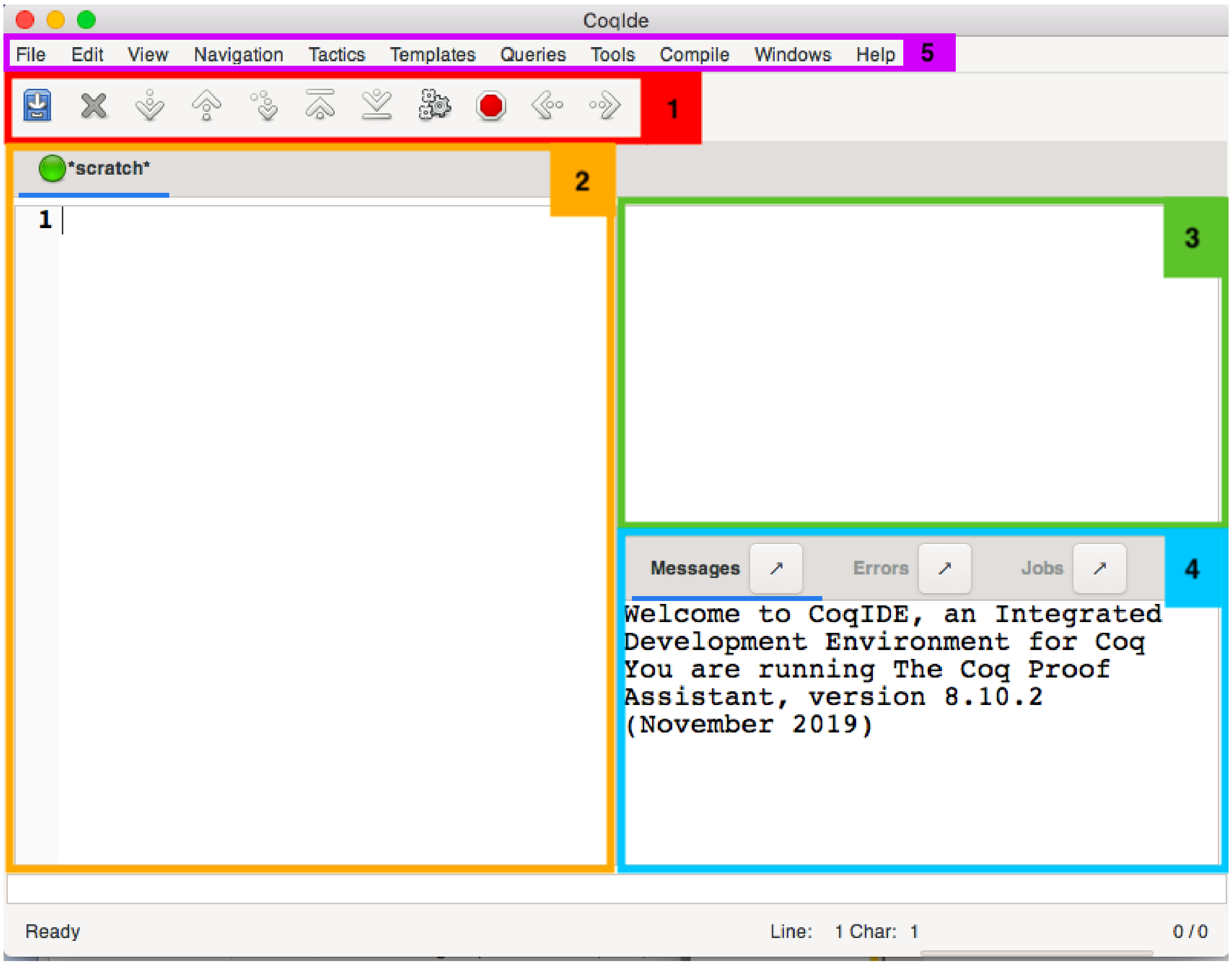
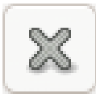


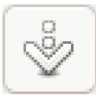
Figure 1: `CoqIDE v8.10.2`: (1) Toolbar. (2) Script Buffer. (3) Goal Window. (4) Message Window. (5) Menu Bar.

3.1 Toolbar for CoqIDE v8.10.2

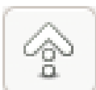
`Save current buffer.` If it hasn't been previously saved, functions as save as; use the extension .v to save as a Coq file.



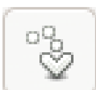
`Close current buffer.` Gives a warning if the file has unsaved changes.



`Forward one command.` Steps forward to evaluate the next command in the current file.



`Backward one command.` Steps backward one command in the file, returns the state to where it was before evaluating that command.

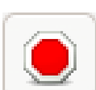


`Go to cursor.` Evaluate all commands in file up to where the cursor currently is.

`Restart Coq.` Returns to the top of the file, where no commands have been evaluated.

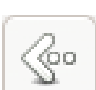
`Go to end.` Evaluate to the bottom of the file. Does not work as well with load commands and require import commands.

`Fully check the document.` Submits proof terms to the Coq kernel for type checking.



`Interrupt computations.` Stops computation at whatever point was reached before pressing the button.

`Next Occurrence.` Goes to the next occurrence of whatever the cursor is currently by. Works well for longer words.



`Previous Occurrence.` Goes to the previous occurrence of whatever the cursor is currently by. Works well for longer words.

3.2 Script Buffer

Here you can type out new definitions and proofs in a new buffer, or open a buffer from a saved file by going to the File tab → Open, then choosing the file. You can edit and save new and existing buffers here.

> This block represents a script buffer in this tutorial.

3.3 Goal Window

Goals to be proven will be displayed here. This window will be empty unless you're in a proof environment; inside the proof environment, it will display what you goal are currently proving, what goals are left to be proved, or that there are no more subgoals and your proof is complete.

> This block represents the goal window in this tutorial.

3.4 Message Window

Any messages resulting from an executed command will be displayed here. Results from queries are also printed out here. Clicking the arrow in the corner of the Messages, Errors, or Jobs tab here will create a separate window for that tab. When the separate window is closed, it will return to the main Coq IDE window.

> This block represents the message window in this tutorial.

3.5 Menu Bar

File

```
File
Edit
View
Navigation
Tactics
Templates
Queries
Tools
Compile
Windows
Help
```

4 The Gallina Specification Language

Gallina is the specification language of Coq. It is a functional language, fairly similar to OCaml or SML if you are familiar with those. It includes things such as pattern matching, let-in definitions, and recursive functions.

The reference manual for the Gallina Language Specification can be found here:
`https://coq.inria.fr/distrib/current/refman/language/gallina-specification-language.html#`

This details the grammar of the language; lexical conventions (i.e. keywords, formatting of identifiers/variables, etc.); the syntax of terms; types; etc; then goes into the grammar of The Vernacular (the language of commands of Gallina). If you are familiar with programming language basics like these, you may find this of interest and it could be beneficial to you; if not, it may be a bit confusing

A few important things to note about the Vernacular of Gallina are that each sentence begins with a capital letter and ends with a dot (i.e. period), and whitespace is used to separate terms but otherwise ignored.

Starting in the Assumptions section, there are some examples mixed in with the formal command definitions and syntax specifications for their use, which may be beneficial if you are struggling with a particular command and need more information beyond what this tutorial provides.

5 The Calculus of Inductive Constructions

The reference manual for the Calculus of Inductive Constructions used by Coq can be found here:
`https://coq.inria.fr/distrib/current/refman/language/cic.html#calculusofinductiveconstructions`

This gives things such as typing rules, conversion rules, subtyping rules, inductive definitions, etc., used by Coq.

This is useful if you'd like a more in-depth perspective on the formal definitions and intuition behind the typing system in Coq - but it is not necessary to know or understand these definitions in great detail to have Coq be of use to you.

This will not be discussed further in this tutorial to keep things simple. Please see the website given above if this is of interest to you.

Part II
Functional Programming in Coq

6 Coq Basics

The language of Coq is functional, and has similarities to OCaml. If you are not familiar with functional programming, this section will help introduce some basic syntax and concepts to get you started. See file "Basics.v"

6.1 Comments

Comments in Coq are surrounded by `(* *)`.

```
(* This is a comment in Coq. *)
```

6.2 Commands

All commands in Coq must end with a period. The various available commands are discussed throughout this document; please see the respective sections for more information on any specific commands.

6.3 Pattern Matching

Pattern matching is a useful way to specify what should occur when you see a specific option of a type. For example, with booleans you have the options of true and false. In general, you have something like this for a single variable:

```
match var with
| opt1 => expr1
| opt2 => expr2
end
```

```
match b with
| true => false
| false => true
end
```

and something like this if you'd like to match over a combination of variables:

```
match (var1, var2) with
| (var1opt1, var2opt1) => expr1
| (var1opt2, var2opt2) => expr2
| ... => ...
| (_ , _ ) => exprX
end
```

The underscore in the last option allows you to specify only the options that you care about at the top, and then for all other cases, do *exprX*. The underscore is not necessary if you've listed all possible options or combinations of options as possible matches.

See example 9.2 for some simple boolean functions using pattern matching.

6.4 Lists

Coq has built in notation for lists that you can use; however, to use the notations, you must make sure to load the `List` library beforehand.

```
Load List.
```

Lists cannot contain elements of different types. If you want to have a list with different types, you must first create a new user-defined type (see defining `Inductive` objects, section 8.3) or use a tuple (see 6.5). From there, you can use the following notations (general case on the left, list of numbers on the right):

```
v1::v2:: ... ::[ ]
```

```
1::2::3::4::5::6::7::8::9::0::[ ]
```

```
[ v1; ...; vN ]
```

```
[ 1; 2; 3; 4; 5; 6; 7; 8; 9; 0 ]
```

The empty list is always denoted as `[ ]`; When using the double colon appended list, you must have the empty list as the right-most element.

You can concatenate two lists (i.e., `l1, l2`) using `l1 ++ l2` or `app l1 l2`. For example,

```
Compute [1] ++[2].
```

```
= [1; 2] : Datatypes.list nat
```

```
Compute app [1] [2; 3].
```

```
= [1; 2; 3] : Datatypes.list nat
```

You can also define your own lists like they are defined in Coq; for example, this defines a list of `nat` numbers:

```
Inductive natlist : Type := | nil | cons (n : nat) (l : natlist).
```

The following functions are defined in the `List` library; for examples of the use of most of these, please see Example 9.5.

`length` :	number of elements in list
`head` :	first element (with default)
`tail` :	all but first element
`app` :	concatenation
`rev` :	reverse
`nth` :	accessing n-th element (with default)
`map` :	applying a function
`flat_map` :	applying a function returning lists
`fold_left` :	iterator (from head to tail)
`fold_right` :	iterator (from tail to head)

6.5 Tuples

A tuple is given by two or more comma separated objects enclosed in parentheses. Tuples can contain items of the same or different types.

```
(v1, ..., vN)
```

```
(1, 2, 3)
```

6.6 Boolean Expressions for Branches

To use boolean expressions on numbers in an `if ... then ... else ...`, make sure you've imported `Arith` and loaded `Bool`:

```
Require Import Arith.
Load Bool.
```

Checking less than:

```
x <? y
```

Checking equal to:

```
x =? y
```

Checking less than or equal to:

```
x <=? y
```

To my knowledge, there is not pre-defined notation in Coq for greater than or not equal operations - however, you can get around this fairly easily by defining the functionality and notations yourself (see Example 9.3).

6.7 Option Types

When using some built-in functions in Coq, you may come across option types. Option types are particularly useful when you want to return an element if it is found, or be told explicitly that it does not exist. In other languages, you would either need to have two separate functions, one to check if it exists and one to obtain the value, or return some sort of 'neutral' value, like -1 or 0, to indicate an item wasn't found. Option types are defined as:

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

and an example of it being used:

```
Fixpoint at_n (n : nat) (l : Datatypes.list nat) : option nat :=
  match l with
    | [ ] => None
    | hd :: tl => if (n =? 0)
                  then Some hd
                  else at_n (n - 1) tl
    end.
```

In this function, we are recursively going through the list to find the item at position n in the list. To do this, at each iteration we are checking if we have found the empty list, if so we return `None`; otherwise we pull apart the head element of the list from the tail of the list, and if n is 0, then we return `Some` with that head element; if n is greater than 0, then we make the recursive call on n `-1` and the tail of the list. The function will execute until we either find the empty list or the element at position n. We can then use pattern matching on the option type to obtain and use the value that was found or do something else having found that the value doesn't exist.

7 Using External Files/Libraries

7.1 Standard Libraries

`Logic`	Classical logic and dependent equality
`Arith`	Basic Peano arithmetic
`PArith`	Basic positive integer arithmetic
`NArith`	Basic binary natural number arithmetic
`ZArith`	Basic relative integer arithmetic
`Numbers`	Various approaches to natural, integer and cyclic numbers (currently axiomatically and on top of 2^{31} binary words)
`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.)
`FSets`	Specification and implementations of finite sets and finite maps (by lists and by AVL trees)
`Reals`	Axiomatization of real numbers (classical, basic functions, integer part, fractional part, limit, derivative, Cauchy series, power series and results,...)
`Relations`	Relations (definitions and basic results)
`Sorting`	Sorted list (basic definitions and heapsort correctness)
`Strings`	8-bits characters and strings
`Wellfounded`	Well-founded relations (basic results)

7.2 Require

To use the standard libraries or other compiled files, you first need to tell the environment that it needs to load the compiled file. `Require` adds the specified module and all of its dependencies to the environment.

```
Require Logic.
```

`Require Import` loads the specified module and its dependences, then imports the contents of the specified module.

```
Require Import Bool.
```

`Require Export` acts like `Require Import`, but will ensure that any module B that uses `Require Import` on the module A that contained the `Require Export` command will import both module A and the one specified in the `Require Export` command.

```
Require Export Bool.
```

7.3 Load

This is used to load a file or library into the current environment. To load one of the standard libraries, you can simply use the `Load` command.

```
Load Arith.
```

However, to load any other existing file, you will likely need to specify where to look for the file; to do this, there is the `Add LoadPath` command. All the commands in the loaded file will be evaluated

```
Add LoadPath "/myDirectory/path/".
Load myFile.
```

8 Defining in Coq

Please see file "Defining.v" to follow along with this code in the CoqIDE.

8.1 Sorts

There are three main `Sorts` for defining types: **Prop**, **Set**, and **Type**.

- **Prop**: This type is for logical propositions.
- **Set**: This type is for small sets, such as booleans (`bool`) and natural numbers (`nat`).
- **Type**: This type can be used for small sets as well as larger sets - it encompasses both **Prop** and **Set**.

These are used in defining Inductive types, as shown in the next section.

8.2 Compute

The command `Compute` evaluates the term that follows it; this is particularly useful to test functions and other elements you have defined. Below are a few examples using some basic arithmetic and boolean equations, others are scattered throughout this tutorial to show the use of various aspects that are discussed.

```
Compute 2 + 4.
```

```
= 6
: Datatypes.nat
```

```
Compute plus 4 2.
```

```
= 6
: Datatypes.nat
```

```
Compute (mult 12 4) * 3.
```

```
= 144
: Datatypes.nat
```

```
Compute true —— false.
```

```
= true
: bool
```

```
Compute false && true.
```

```
= false
: bool
```

8.3 Inductive

The command `Inductive` is used to define simple inductive types and the constructors used in the type. The name is placed directly after the keyword `Inductive`, when a colon followed by the `Sort` (discussed in the previous section) of the inductive type you are defining. This is followed by := and the constructors (i.e. elements) included in that type. For example, boolean values are defined in Coq as follows:

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

```
bool is defined
bool_rect is defined
bool_ind is defined
bool_rec is defined
```

and `nat` numbers are defined as:

```
Inductive nat : Set :=
    | O : nat
    | S : nat − > nat.
```

```
nat is defined
nat_rect is defined
nat_ind is defined
nat_rec is defined
```

You can see what all the definitions it created are by using `Print`:

```
Print nat.
```

```
Inductive nat : Set := O : nat | S : nat − > nat
```

```
Print nat_rect.
```

```
nat_rect =
fun (P : nat − > Type) (f : P O) (f0 : forall n : nat, P n − > P (S n)) =>
fix F (n : nat) : P n := match n as n0 return (P n0) with
            | O => f
            | S n0 => f0 n0 (F n0)
            end
      : forall P : nat − > Type, P O − > (forall n : nat, P n − > P (S n)) − > forall n : nat, P n

Argument scopes are [function_scope _ function_scope _ ]
```

```
Print nat_ind.
```

```
nat_rect =
fun (P : nat − > Prop) (f : P O) (f0 : forall n : nat, P n − > P (S n)) =>
fix F (n : nat) : P n := match n as n0 return (P n0) with
            | O => f
            | S n0 => f0 n0 (F n0)
            end
      : forall P : nat − > Prop, P O − > (forall n : nat, P n − > P (S n)) − > forall n : nat, P n

Argument scopes are [function_scope _ function_scope _ ]
```

```
Print nat_rec.
```

```
nat_rec =
fun P : nat − > Set => nat_rect P
      : forall P : nat -> Set, P O − > (forall n : nat, P n − > P (S n)) − > forall n : nat, P n

Argument scopes are [function_scope _ function_scope _ ]
```

You can use these properties of what you've defined in proofs.

Similarly, you can define days of the week:

```
Inductive day: Type :=
  | monday : day
  | tuesday : day
  | wednesday : day
  | thursday : day
  | friday : day
  | saturday : day
  | sunday : day.
```

```
day is defined
day_rect is defined
day_ind is defined
day_rec is defined
```

or lists of natural numbers:

```
Inductive natlist Type :=
  | nil
  | cons (n:Datatypes.nat) (l:natlist).
```

```
list is defined
list_rect is defined
list_ind is defined
list_rec is defined
```

It is also possible to define polymorphic lists:

```
Inductive list (A: Set) : Set :=
  | nil : list A
  | cons : A − > list A.
```

```
list is defined
list_rect is defined
list_ind is defined
list_rec is defined
```

(polymorphic lists will not be used in the remainder of the tutorial - it is just an example of a more complex construct that can be defined in COQ).

8.4 Definition

The command `Definition` is used to bind an name to some term. The name is always placed directly after the keyword `Definition`, and the term to bind to the name is given after :=. For example, we can give the name x a simple value of 4:

```
Definition x := 4.
```

```
x is defined
```

or we can use this to define functions, such as the following simple function (using the previous weekday inductive type definition), taking a weekday as input and giving back the weekday as output. Here, we are specifying the parameter d of type day must be given to the function. When giving a parameter, you give the ($paramName : type$) as in ($d : day$). The parameters are then followed by the return type, as in $: returnType$. This is shown in the following example.

```
Definition next_weekday (d:day) : day :=
  match d with
    | monday => tuesday
    | tuesday => wednesday
    | wednesday => thursday
    | thursday => friday
    | friday => monday
    | saturday => monday
    | sunday => monday
  end.
```

```
next_weekday is defined
```

Another simple example function definition (using the previous `nat` number inductive type definition) is taking a `nat` number and returning that result of adding 2 to that `nat` number:

```
Definition plus2 (n:nat) : nat :=
  match n with
    | O => S (S O)
    | _ => S (S n)
  end.
```

```
plus2 is defined
```

You can also give multiple parameters, as shown in the example below using 3 parameters:

```
Definition choose1 (b: bool) (n1: Datatypes.nat) (n2: Datatypes.nat) :
Datatypes.nat :=
  match b with
    | true => n1
    | false => n2
  end.
```

```
choose1 is defined
```

We have to specify that we would like to use `Datatypes.nat` as our type in order to use natural numbers (i.e. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9), as we have not specified how to interpret these numbers from our definition of nat (you can do this using the `Notation` command - using this command will be discussed in the following subsection 8.5). Alternately, we can define the same function without giving the types of the parameters (however, it is always best practice to declare the types of all parameters, to ensure they are interpreted as you expect them to be). When you do not specify the types of parameters, the parentheses around the parameter names are optional.

```
Definition choose1' b n1 n2 : Datatypes.nat :=
  match b with
    | true => n1
    | false => n2
  end.
```

```
choose1' is defined
```

Both *choose1* and *choose1′* have the same functionality.

8.5 Notation

The command `Notation` is used to define a short-hand way of writing a concept we have previously defined - for example, take our list definition from subsection 8.3. We can then make the shorthand notation where [ ] refers to *nil*, or the empty list, and $x :: xs$ refers to ($cons\ x\ xs$), or appending some number x to some list xs.

```
Notation "[ ]" := nil.
Notation "x :: xs" := (cons x xs).
```

```
Setting notation at level 0.
```

Now we are able to write lists more simply (shown compared to the identical list using *cons* and *nil*):

```
Compute 3::2::1::[ ].
Compute (cons 3 (cons 2 (cons 1 nil))).
```

```
= 3 :: 2 :: 1 :: [ ]
: natlist
```

This can be used to define any number of concepts that you may desire, including notations for the boolean expressions for checking where some number is greater than some other number (which are not natively defined, as discussed in subsection 6.6). These notations and definitions are shown in example 9.3.

8.6 Fixpoint

The command `Fixpoint` is used to define recursive functions using a fixed point construction. These functions use pattern matching over inductive objects, and must have a decreasing argument to ensure termination. The decreasing argument is best understood through examples, but can be thought of as the object that is controlling the recursion (and, if using a `match` x `with ... end` statement, is likely x). It is best practice to explicitly declare the decreasing argument in the function definition using { `struct id` }, though it can be left implicit. Coq will give an error if there is an issue with the decreasing argument of a `Fixpoint` definition.

The following function *search* recursively searches the given list l to see if it contains the number n, and returns a `bool` (i.e., `true` or `false`). Here, as shown in the function definition and the messages, the decreasing argument is the first argument, l. This function first checks to see if the list l is empty, returning `false` if it is. Otherwise, the function will break l into the first (i.e., head) element hd and the remaining (i.e., tail) list; then check to see if hd is equal to n, returning `true` if this holds, if not, it will recursively call itself to check the remaining list tl to see if n is present.

```
Fixpoint search
(l : natlist) (n : Datatypes.nat) { struct l} : bool :=
   match l with
      | [ ] => false
      | hd::tl =>
           if (n =? hd)
           then true
           else search tl n
   end.
```

```
search is defined
search is recursively defined (decreasing on 1st argument)
```

The following is essentially the same function, with the order of arguments switched and the decreasing argument left implicit. Coq will still recognize that the l is the decreasing argument.

```
Fixpoint search2
(n : Datatypes.nat) (l : natlist) : bool :=
   match l with
      | [ ] => false
      | hd::tl =>
           if (n =? hd)
           then true
           else search2 n tl
   end.
```

```
search2 is defined
search2 is recursively defined (decreasing on 2nd argument)
```

9 Examples: Programming in Coq

9.1 Cards

Please see the "cards.v" file to follow along with this example.
This is a simple example defining suits and values for cards, and what a valid card is. In the function `check_num`, we check if the number given is less than 11 and greater than 1. In function `is_valid_card`, we pattern match against various types of cards. Here, the given card is represented by the variable x. The type card is defined to be `c (s: suit) (v: val)`, and we know that an `s` of type suit can only be one of the valid suits we defined, so we can use some variable, say `q`, to represent allowing any suit type; the part we need to ensure is valid is the value given to the suit, since a valid card can only have a value of an ace, king, queen, jack, or number values 2 thru 10.

```
Load List.
Load Bool.

Inductive suit : Type := | heart | diamond | spade | club.
Inductive val : Type := | ace | king | queen | jack | num (n: nat).
Inductive card : Type := | c (s: suit) (v: val).

(* Card: 2 of Spades *)
Check (c spade (num 2)).

(* List of suits *)
Check (ace::king::[ ]).

(* List of Cards *)
Check [c heart ace; c diamond king; c spade queen].

Definition check_num (x: nat) : bool :=
  if (x <? 11)
  then if (1 <? x)
    then true
    else false
  else false.

Definition is_valid_card (x: card) : bool :=
  match x with
  | c q ace => true
  | c q king => true
  | c q queen => true
  | c q jack => true
  | c q (num n) =>
      if (check_num n)
      then true
      else false
  end.

(* Ace of Hearts is a valid card *)
Compute is_valid_card (c heart ace).

(* 11 of Diamonds is NOT a valid card *)
Compute is_valid_card (c diamond (num 11)).
```

9.2 Boolean Operations

Please see the "bool.v" file to follow along with this example.
This is a simple example of the definition of booleans and some boolean operations.

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

Definition not (b : bool) : bool :=
  match b with
  | true => false
  | false => true
  end.

Definition and (b1 b2 : bool) : bool :=
  match (b1, b2) with
  | (true, true) => true
  | (_ , _ ) => false
  end.

Definition or (b1 b2 : bool) : bool :=
  match (b1, b2) with
  | (false, false) => false
  | (_ , _ ) => true
  end.

Definition xor (b1 b2 : bool) : bool :=
  match (b1, b2) with
  | (true, false) => true
  | (false, true) => true
  | (_ , _ ) => false
  end.
```

9.3 Boolean Expressions for Branches

Please see the "bool_expr.v" file to follow along with this example.
This is a simple example defining less than, equal to, and less than or equal to operations over numbers.

```
Require Import Arith.
Load Bool.

Definition lt (n m : nat) : bool :=
    if (n <? m)
    then true
    else false.
Notation "n < m" := (lt n m).

Definition gt (n m : nat) : bool :=
    if (m <? n)
    then true
    else false.
Notation "n > m" := (gt n m).

Definition eq (n m : nat) : bool :=
    if (n =? m)
    then true
    else false.
Notation "n = m" := (eq n m).

Definition neq (n m : nat) : bool :=
    if (n =? m)
    then false
    else true.

Definition lteq (n m : nat) : bool :=
    if (n <=? m)
    then true
    else false.
Notation "n <= m" := (lteq n m).

Definition gteq (n m : nat) : bool :=
    if (m <=? n)
    then true
    else false.
Notation "n >= m" := (gteq n m).
```

9.4 Days of the Week

Please see file "days.v" to follow along with this example.

This example is fairly simple, giving some basic definitions and functions, and showing a use of a tuple in Coq.

First, we define what a day is using the following definition:

```
Inductive day: Type :=
  | monday : day
  | tuesday : day
  | wednesday : day
  | thursday : day
  | friday : day
  | saturday : day
  | sunday : day.
```

Then we define a simple function to compute what the next weekday is after a given day. This takes a day as input, and gives back what the next weekday would be.

```
Definition next_weekday (d:day): day :=
  match d with
    | monday => tuesday
    | tuesday => wednesday
    | wednesday => thursday
    | thursday => friday
    | friday => monday
    | saturday => monday
    | sunday => monday
  end.
```

We can perform some simple computations to check the correctness of our function:

```
Compute (next_weekday friday).
```

```
= monday : day
```

```
Compute next_weekday (next_weekday friday).
```

```
= tuesday : day
```

To make this a bit more interesting, we can also define some other things, such as sports:

```
Inductive sport: Type :=
  | tennis : sport
  | basketball : sport
  | baseball : sport
  | cricket : sport
  | football : sport
  | dance : sport
  | gymnastics : sport
  | run : sport
  | rugby : sport
  | weights : sport
  | swim : sport.
```

and activities:

```
Inductive activity: Type :=
  | class : activity
  | lab : activity
  | meeting : activity
  | seminar : activity
  | volunteer : activity
  | club : activity
  | work : activity.
```

Now, it is possible to create a function that returns a schedule for each day, consisting of a tuple of an activity and a sport.

```
Definition daily_schedule (d:day): activity * sport :=
  match d with
    | monday => (meeting, run)
    | tuesday => (class, tennis)
    | wednesday => (seminar, baseball)
    | thursday => (class, rugby)
    | friday => (lab, dance)
    | saturday => (club, swim)
    | sunday => (volunteer, basketball)
  end.
```

Of course, it is also possible to create more complex schedules, but this should give you an the basic idea of how to do that.

9.5 List

A simple example to demonstrate some uses of the built-in lists and list functions, and define a search function for lists of nat numbers. To use the notations and functions, you'll want to load List and open the list_scope so everything works properly.

```
Require Import Arith.
Load List.
Open Scope list_scope.

Fixpoint search (l : Datatypes.list nat) (n: nat) : bool :=
  match l with
    | [ ] => false
    | hd :: tl =>
        if (n =? hd)
        then true
        else search tl n
  end.
```

Uses of the list functions and results:

Compute search [1;3;6;8] 5.	= false : bool
Compute length [1;2;3].	= 3 : nat
Compute head [2;4;6].	= Some 2 : option nat
Compute tail [3;6;9].	= [6; 9] : Datatypes.list nat
Compute app [1;2] [3;4].	= [1; 2; 3; 4] : Datatypes.list nat
Compute [1;2] ++ [3;4].	= [1; 2; 3; 4] : Datatypes.list nat
Compute 1::[2;3;5;7].	= [1; 2; 3; 5; 7] : Datatypes.list nat
Compute rev [9;7;5;3;1].	= [1; 3; 5; 7; 9] : Datatypes.list nat
Compute nth 0 [2;5;7;9].	= fun _ : nat => 2 : nat -> nat

```
Compute nth 4 [2;5;7;9].
(* Not found *)
```

```
= fun default : nat => default
: nat -> nat
```

```
Compute map (Nat.mul 3) [1;2;3].
```

```
= [3; 6; 9] : Datatypes.list nat
```

Part III
Proving Properties in Coq

10 Proof Environment

10.1 Assertions

To switch on the proof environment, you first need to use one of the keywords for an assertion. It does not matter which one you choose, all have the same behavior - it is more a matter of personal preference.
These keywords are:

Theorem, Lemma, Corollary, Proposition, Fact, Goal, Example, Remark

After using one of the keywords, you will give your assertion a name, followed by a semi-colon, then write out the body of your assertion, followed by a period.
The theorem template (Templates tab → Theorem) is shown below:

```
Theorem new_theorem : .
Proof.

Qed.
```

10.2 Proof Environment Commands

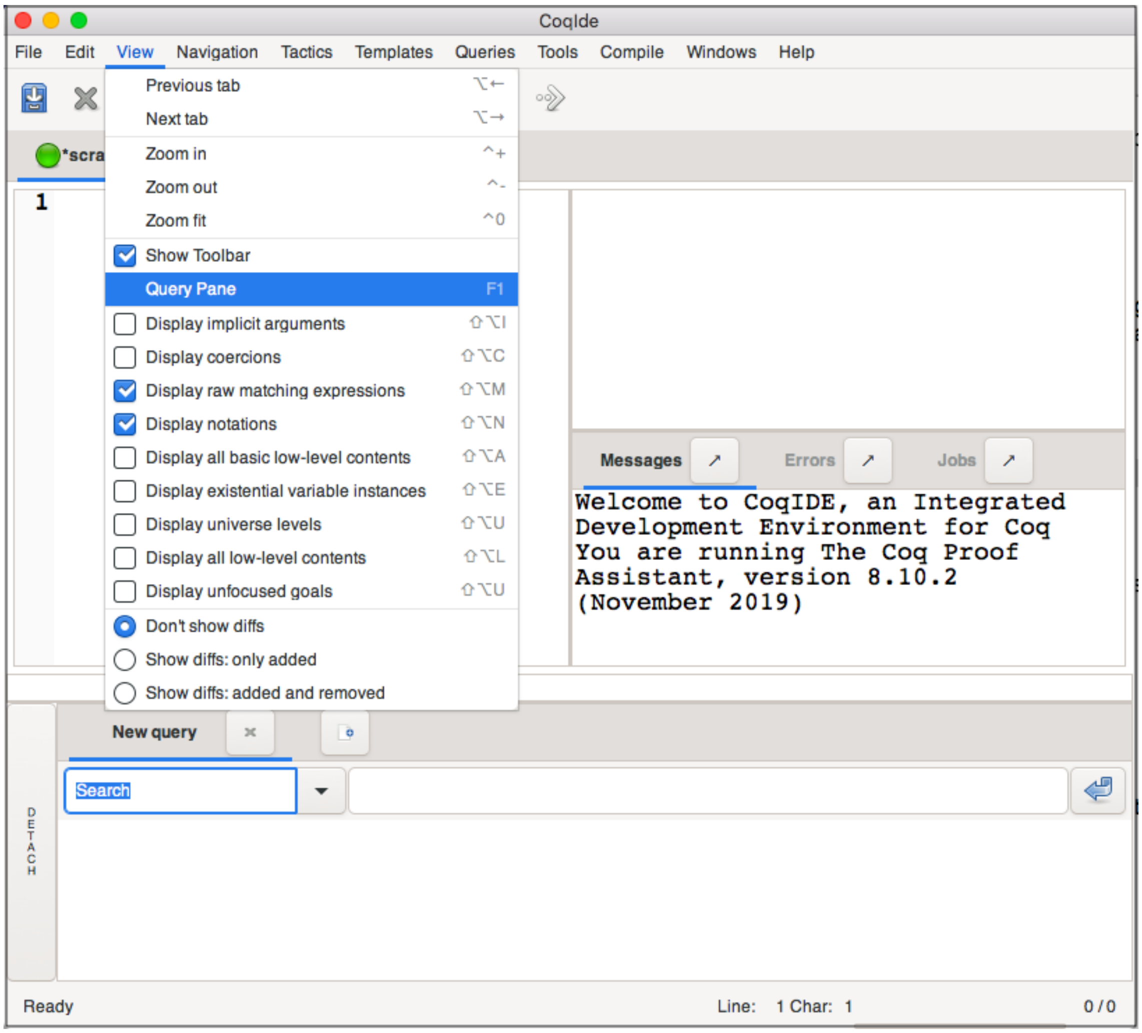
Proof.	Begins the proof.
Qed.	Ends and saves the proof; will fail if the assertion is not fully proven.
Admitted.	Gives up the current proof; declares the initial goal as an axiom that can be used in other proofs (but will need to be fully proven later).
Abort.	Cancels the current proof development.

11 Queries

You can insert queries into your file (despite the Coq IDE giving a warning), or you can use the query pane. The results of a query will appear in the messages pane. There are many types of queries, including the ones described below. See file "Queries.v" to follow along.

11.1 Query Pane

To open the query pane, either press F1 or go to the View tab, then down to the Query Pane option. The query pane is more convenient when you're in the middle of a proof but need to look up something; however, when doing multiple queries from the query pane, the results will show up in the messages tab without clearing the results of previous queries.



You can detach the query pane into its own window by pressing the Detach button along the lefthand side of the query pane.

11.2 Search

Searches the environment for the given name, then displays the name and type of all objects that contains the name. This is useful to find out information about libraries and pre-defined theorems.

```
Search plus.
```

```
plus_O_n: forall n : nat, 0 + n = n
plus_n_O: forall n : nat, n = n + 0
plus_n_Sm: forall n m : nat, S (n + m) = n + S m
plus_Sn_m: forall n m : nat, S n + m = S (n + m)
mult_n_Sm: forall n m : nat, n * m + n = n * S m
f_equal2_plus: forall x1 y1 x2 y2 : nat, x1 = y1 -> x2 = y2 -> x1 + x2 = y1 + y2
nat_rect_plus:
   forall (n m : nat) (A : Type) (f : A -> A) (x : A),
   nat_rect (fun _ : nat => A) x (fun _ : nat => f) (n + m) =
   nat_rect (fun _ : nat => A) (nat_rect (fun _ : nat => A) x (fun _ : nat => f) m)
     (fun _ : nat => f) n
```

Searching multiple names obtains more specific results.

```
Search plus 0.
```

```
plus_O_n: forall n : nat, 0 + n = n
plus_n_O: forall n : nat, n = n + 0
```

If you haven't loaded the library corresponding to what you are searching, you will get fewer results; you may need to load the library if you can't find what you need. For example,

```
Search plus Nat.odd.
```

yields no results when the Arith library hasn't been added, but once we have imported the library we receive five results.

```
Require Import Arith.
Search plus Nat.odd.
```

```
Nat.odd_add_even: forall n m : nat, Nat.Even m -> Nat.odd (n + m) = Nat.odd n
Nat.odd_add: forall n m : nat, Nat.odd (n + m) = xorb (Nat.odd n) (Nat.odd m)
Nat.odd_add_mul_even: forall n m p : nat, Nat.Even m -> Nat.odd (n + m * p) = Nat.odd n
Nat.odd_add_mul_2: forall n m : nat, Nat.odd (n + 2 * m) = Nat.odd n
Nat.div2_odd: forall a : nat, a = 2 * Nat.div2 a + Nat.b2n (Nat.odd a)
```

You can also search for patterns: enclose the pattern in parentheses, and use an underscore for arbitrary terms.

```
Search (~ _ <-> _ ).
```

```
neg_false: forall A : Prop, ~ A <-> (A <-> False)
not_iff_compat: forall A B : Prop, A <-> B -> ~ A <-> ~ B
```

11.3 SearchRewrite

Searches the environment for a statement with an equality that contains the given pattern on at least one side.

```
SearchRewrite (_ * _ / _ ).
```

```
Nat.div_mul: forall a b : nat, b <> 0 -> a * b / b = a
Nat.divide_div_mul_exact: forall a b c : nat, b <> 0 -> Nat.divide b a -> c * a / b = c * (a / b)
Nat.div_mul_cancel_l: forall a b c : nat, b <> 0 -> c <> 0 -> c * a / (c * b) = a / b
Nat.div_mul_cancel_r: forall a b c : nat, b <> 0 -> c <> 0 -> a * c / (b * c) = a / b
Nat.lcm_equiv1: forall a b : nat, Nat.gcd a b <> 0 -> a * (b / Nat.gcd a b) = a * b / Nat.gcd a b
Nat.lcm_equiv2: forall a b : nat, Nat.gcd a b <> 0 -> a / Nat.gcd a b * b = a * b / Nat.gcd a b
```

11.4 Check

Displays the type of the given term.

```
Check 0.
```

```
0 : nat
```

```
Check nat.
```

```
nat : Set
```

```
Check plus.
```

```
Nat.add : nat -> nat -> nat
```

11.5 About

Displays information about the object with the given name (i.e. its kind, type, arguments).

```
About plus.
```

```
Notation plus := Init.Nat.add
Expands to: Notation Coq.Init.Peano.plus
```

11.6 Locate

Displays the full name of the given term and what Coq module it is defined in.

```
Locate plus.
```

```
Notation Coq.Init.Peano.plus
```

```
Locate nat.
```

```
Inductive Coq.Init.Datatypes.nat
```

11.7 Print

Displays information about the given name.

```
Print plus.
```

```
Notation plus := Init.Nat.add
```

It is possible to see what is currently loaded and imported in the environment with the following:

```
Print Libraries.
```

```
Loaded and imported library files:
  Coq.Init.Notations
  Coq.Init.Logic
  Coq.Init.Logic_Type
  Coq.Init.Datatypes
  Coq.Init.Specif
  Coq.Init.Peano
  Coq.Init.Wf
  Coq.Init.Tactics
  Coq.Init.Tauto
  Coq.Init.Prelude
Loaded and not imported library files:
  Coq.Init.Nat
```

11.8 Print Assumptions

Displays all assumptions depended upon by the object of the given name.

```
Print Assumptions plus.
```

```
Closed under the global context
```

12 Proof Tactics

This section contains brief descriptions of some commonly used proof tactics. For the full breakdown of all Coq tactics, please refer to the Tactic Index of the Coq documentation that can be found here: `https://coq.inria.fr/distrib/current/refman/coq-tacindex.html`.

12.1 simpl

Simplify both sides of an equation.
Examples using this tactic: 13.4

12.2 reflexivity

Check if both sides of an equation are equal. Will do more simplifications that simpl; tries unfolding and expanding definitions.
Examples using this tactic: 13.4

12.3 trivial

A restriction of auto that is not recursive. Tries simple hints, like solving trivial equalities such as x=x.
Examples using this tactic: 13.4

12.4 auto

First attempts to use assumption, then uses intros and generates any hypotheses to use as hints and attempt to apply.
Examples using this tactic:

12.5 ring

Very useful when proving over numbers. Solves equations upon polynomial expressions of a ring structure. Normalizes and compares results. Uses properties like associativity, commutativity, distributivity, and constant propagation.
Examples using this tactic: 13.4

12.6 discriminate

Proves any goal in an assumption stating two structurally different terms of an inductive set are equal. For example, S (S O) = S O.
Examples using this tactic:

12.7 assumption

Looks in the local context for the hypothesis. The type must be convertible to the goal.
Examples using this tactic:

12.8 intros

Introduces variables from the goal into the proof environment for use.
Examples using this tactic: 13.4

12.9 contradiction

Attempts to find a hypothesis equivalent to:

- an empty inductive type (i.e. false)
- the negation of a single inductive type (i.e. true, x=x)
- two contradictory hypotheses

Examples using this tactic:

12.10 induction

The object must be of an inductive type to use induction. This generates subgoals and an induction hypothesis.
Examples using this tactic:

12.11 functional induction

Very useful for inductive/recursive functions. Performs case analysis and induction on the definition of a function. To use functional induction, you need to make sure to require and load FunInd, and then define the functional induction scheme.

```
Require FunInd.
Load FunInd.

Functional Scheme name_ind := Induction for name Sort Prop.
```

Examples using this tactic: 13.4

12.12 destruct

Creates subgoals from a more complex goal. Subgoals must be proven separately. Can be used with any inductively defined type. Can be used nested if a generated subgoal needs to be broken up further. Allows you to specify the names of variables to be used in proving the subgoals.
Examples using this tactic:

12.13 rewrite

A way to apply previously defined assumptions. Can use arrows $<-$, $->$ to give directionality for the application. Example 13.4 demonstrates the use of arrows to give directionality to rewrite.
Examples using this tactic: 13.4

12.14 apply

Attempts to match the current goal against the conclusion of the given term.
Examples using this tactic:

13 Examples: Proving in Coq

13.1 Days of the Week

Please see file "days.v" to follow along with this example.

This example is fairly simple, giving some basic definitions and functions, and showing a use of a tuple in Coq.

First, we define what a day is using the following definition:

```
Inductive day: Type :=
    | monday : day
    | tuesday : day
    | wednesday : day
    | thursday : day
    | friday : day
    | saturday : day
    | sunday : day.
```

Then we define a simple function to compute what the next weekday is after a given day. This takes a day as input, and gives back what the next weekday would be.

```
Definition next_weekday (d:day): day :=
    match d with
      | monday => tuesday
      | tuesday => wednesday
      | wednesday => thursday
      | thursday => friday
      | friday => monday
      | saturday => monday
      | sunday => monday
    end.
```

We can perform some simple computations to check the correctness of our function:

```
Compute (next_weekday friday).
```

```
= monday : day
```

```
Compute next_weekday (next_weekday friday).
```

```
= tuesday : day
```

To make this a bit more interesting, we can also define some other things, such as sports:

```
Inductive sport: Type :=
    | tennis : sport
    | basketball : sport
    | baseball : sport
    | cricket : sport
    | football : sport
    | dance : sport
    | gymnastics : sport
    | run : sport
    | rugby : sport
    | weights : sport
    | swim : sport.
```

and activities:

```
Inductive activity: Type :=
  | class : activity
  | lab : activity
  | meeting : activity
  | seminar : activity
  | volunteer : activity
  | club : activity
  | work : activity.
```

Now, it is possible to create a function that returns a schedule for each day, consisting of a tuple of an activity and a sport.

```
Definition daily_schedule (d:day): activity * sport :=
  match d with
    | monday => (meeting, run)
    | tuesday => (class, tennis)
    | wednesday => (seminar, baseball)
    | thursday => (class, rugby)
    | friday => (lab, dance)
    | saturday => (club, swim)
    | sunday => (volunteer, basketball)
  end.
```

Of course, it is also possible to create more complex schedules, but this should give you an the basic idea of how to do that.

13.2 List Rev

See Coq file “rev_list.v” to follow along with this example.

13.3 Factorial

13.4 Sum to n

This example defines a recursive function that takes a number n and returns the sum of the numbers from one to n. It then proves that for all nat numbers, 2 * sum n = (1 + n) * n. This assertion is equivalent to sum n = ((1 + n) * n) / 2; however, it is much easier to prove when we do not involve division. First, a longer proof using properties of addition and multiplication will be shown, then the shorter version that applies the ring tactic to take care of the application of many of these properties for us.

```
Load Arith.

Fixpoint sum (n : nat) : nat :=
  match n with
  | O => 0
  | S p => (S p) + (sum p)
  end.
```

In this example, we will use functional induction to help us successfully complete our proof over the recursive function sum. To do this, we must first make sure to require and load FunInd, and then define the functional induction scheme.

13.5 Sum of Squares

14 Troubleshooting

14.1 Empty IDE

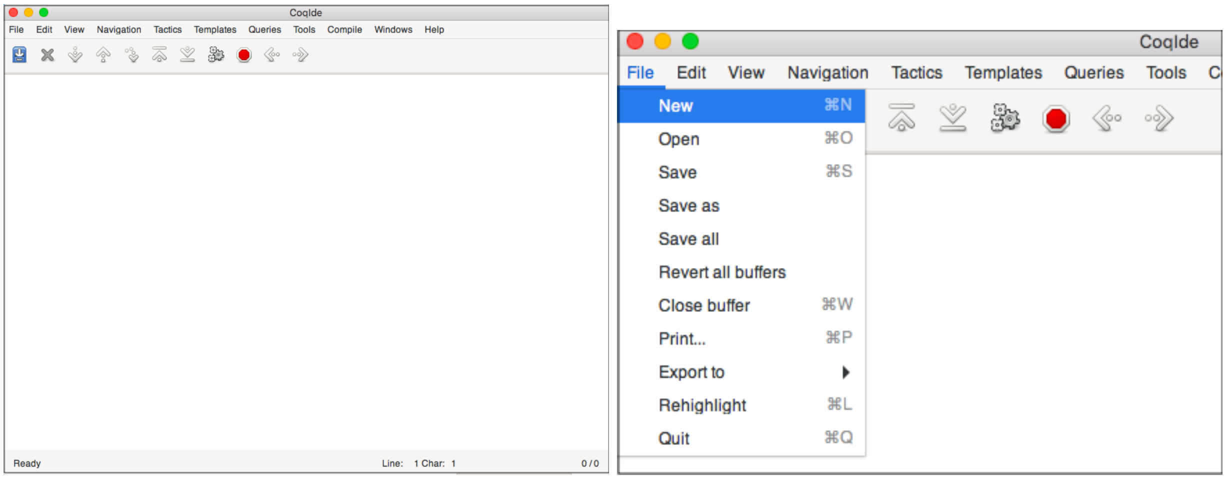


Figure 2: CoqIDE v8.10.2 (a) empty IDE screen (b) getting new scratch script buffer open

Go to `File` then `New` to get a new scratch script buffer open if you've accidentally closed out all of your script buffers. Alternately, you can go to `File` then `Open` if you'd like to open a saved file.

14.2 Can't close `Load File` window

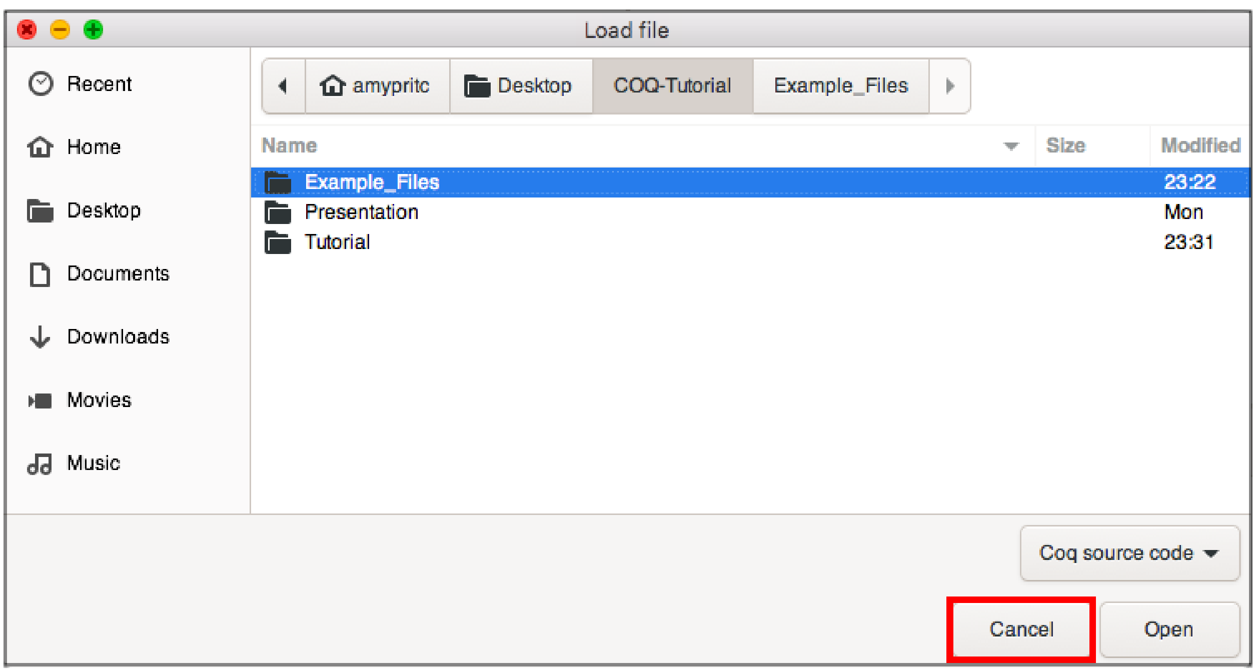


Figure 3: CoqIDE v8.10.2 Load File window

If the `Load File` window won't close using the red X in the upper left-hand corner, use the `Cancel` button in the lower right (outlined in red).

14.3 File didn't save with the `.v` extension

When saving a new buffer, you need to ensure you add the `.v` extension to the file name. If you forgot to do this, you can either use `File` then `Save as` (then reopen the correctly named program) or modify the name outside of the CoqIDE (i.e. command line or file explorer such as Finder for MacOS).

14.4 Can't get pre-defined datatype to work

Try using the full name of the type - i.e.
`nat`: use `Datatype.nat`
`list`: use `Datatype.list`

14.5 Copy/paste program from a file into CoqIDE - program not working

Copying from a file and pasting into the CoqIDE can cause some issues. The most common one I have found is that any use of the underscore (i.e., _) tends to get erased. If you are looking to try out code from this tutorial, the best way to do so is to use the provided example files, as these have all been tested to ensure they work properly in the CoqIDE.

14.6 CoqIDE running very slow

I do not have a solution to this, but have seen the CoqIDE run very slowly on a VM running linux (to the point that it was nearly impossible to use). Installing on Windows and Mac with version 8.10.2 should work well with a pretty good response time (load libraries will take slightly longer than just stepping through code, but is still within reason). If you run into this issue, try reinstalling or heading over to the Coq website (`https://coq.inria.fr`) for assistance.

15 References

Coq Proof Assistant Webpage
`https://coq.inria.fr`

Coq Documentation
`https://coq.inria.fr/distrib/current/refman/`

Christine Paulin-Mohring's course notes on Coq
`https://www.lri.fr/~paulin/LASER/course-notes.pdf`

DeepSpec Summer School COQ Intensive July 2017
`https://deepspec.org/event/dsss17/coq_intensive.html`

Software Foundations Books
`https://softwarefoundations.cis.upenn.edu`

Induction in Coq
`http://www.cs.cornell.edu/courses/cs3110/2018sp/l/22-coq-induction/notes.html`

CompCert: Verified C Compiler
`http://compcert.inria.fr/motivations.html`

CertiCrypt: Computer-Aided Cryptographic Proofs in Coq
`http://certicrypt.gforge.inria.fr`

OCaml Language
`https://ocaml.org`
`https://en.wikipedia.org/wiki/OCaml`

SML Language
`https://en.wikipedia.org/wiki/Standard_ML`
`http://sml-family.org`
`https://www.smlnj.org`