Types Summer School 2007
Coq-lab

Christine Paulin-Mohring and Jean-Christophe Filliâtre

Contents

1 Using the Coq Proof Assistant

1.1 Launching Coq

The command `coqide` starts the Coq graphical interface. It is composed of three windows. The left one contains the user input script. The right-top window contains the current goal. The right-bottom window contains the output of commands. All commands ends with a dot. There is also a pure ASCII toplevel (`coqtop`) and a batch compiler (`coqc`, see Section 3.1). The Coq reference manual is available online at

`coq.inria.fr/V8.1/refman/index.html`

1.2 Syntax of Terms

We assume that the syntax of terms is already known (see Christine's lecture).

1.3 The Coq environment

The following interactive commands are useful to navigate through libraries when doing proofs. They can be executed from the `coqide` Query window (use the menu `Queries` or `Windows/Show Query Window`).

- `Inspect` *num*: displays the type of the last *num* declarations. A declaration is either the introduction of a new identifier or the loading of a module.

```
Coq < Definition x := 0.
Coq < Inductive y : Set := C : y.

Coq < Inspect 5.
x : nat
Inductive y : Set :=  C : y
y_rect : forall P : y -> Type, P C -> forall y : y, P y
y_ind : forall P : y -> Prop, P C -> forall y : y, P y
y_rec : forall P : y -> Set, P C -> forall y : y, P y
```

- `Search` *name*: displays all declaration *id : type* where *name* is the head constant of *type*, *i.e. type* is of the form

$$\texttt{forall}\ (x_1 : A_1)\ \ldots\ (x_n : A_n),\ \textit{name}\ t_1 \ldots t_p$$

```
Coq < Search not.
sym_not_eq: forall (A : Type) (x y : A), x <> y -> y <> x
sym_not_equal: forall (A : Type) (x y : A), x <> y -> y <> x
not_eq_S: forall n m : nat, n <> m -> S n <> S m
O_S: forall n : nat, 0 <> S n
n_Sn: forall n : nat, n <> S n
```

  Application: Find all lemmas about conjunction `and`, disjunction `or`, equality `eq` and order relation `le`.

- `Print` *name*: prints the definition of *name* together with its type.

  Application: Find the definitions of type `nat`, order relations `le` and `lt` and of the proofs `eq_S` and `f_equal`. Find the definitions of operations `plus` and `mult`. Note that the latter are printed as infix symbols + and $\star$.

- `Check` *term*: checks if *term* can be typed and displays its type. If *term* is an identifier, it is another way to get its type.

```
Coq < Check (fun x:nat => x + 0).
fun x : nat => x + 0
     : nat -> nat
```

  Application: What is proved by lemma `nat_case`? Are the following terms well-typed: `(plus O (S O)), (plus true false)`?

- `Require Export` *name*: checks if module *name* is already present in the environment. If not, and if a file *name.vo* occurs in the loadpath, then it is loaded and opened (its contents is revealed).

  The set of loaded modules and the loadpath can be displayed with the commands `Print Modules` and `Print LoadPath`. The default loadpath is the set of all subdirectories of the Coq standard library.

  The libraries related to natural numbers arithmetic are `Le, Lt, Gt, Plus, Mult, Between`. They are gathered in a single module `Arith` in such a way that the command `Require Export Arith` loads and opens all these modules.

```
Coq < Require Export Arith.
```

  Applications: Display the loadpath. Load one of the arithmetic modules and search for theorems about relation `le`.

- `Reset` *name*: Restore the system as it was before the declaration of *name*. It is not useful in `coqide` where navigation through the input script is provided by menus and key shortcuts.

1.4 Parameters and Definitions

1.4.1 Parameters

It is possible to introduce parameters for the theory under development. The syntax is

{`Hypothesis | Variable | Axiom | Parameter`} *name : type*

where *name* is the name of the hypothesis or variable to introduce and *type* its type. The four variants `Hypothesis`, `Variable`, `Axiom` and `Parameter` are all equivalent; they are provided for the (mathematical) clarity of the development. Several variables of the same type can be introduced with a single command, using the variants `Variables` and `Hypotheses` and a blank-separated list of names.

For instance, we can introduce a type `A` and two variables of this type using the commands:

```
Coq < Variable A : Set.
Coq < Variables x y : A.
```

1.4.2 Definitions

The usual way to introduce a new definition is

`Definition` *name* `:=` *term* `:` *type*

The identifier *name* is then an abbreviation for the term *term*. The type *type* is optional (and inferred when omitted).

Example. The square function can be defined as follows:

```
Coq < Definition square := fun x:nat => x * x.
```

or equivalently as follows:

```
Coq < Definition square (x:nat) : nat := x * x.
```

1.5 Inductive Declarations

1.5.1 Inductive Data Types

Possibly recursive data types can be declared as a set of constructors. Each constructor is associated to a type which describes its *arity*. There are some syntactic restrictions over arities.

The syntax of inductive declarations is:

`Inductive` *name* `: Set :=` $c_1 : C_1 \mid \ldots \mid c_n : C_n$

where *name* is the name of the type to be defined, c_i the names of the constructors and each C_i the type of constructor c_i.

Example. The data type of natural numbers can be declared as follows:

```
Coq < Inductive nat : Set :=
Coq < | O : nat
Coq < | S : nat -> nat.
```

Note that constructor names must be valid identifiers and thus `O` is the capital character and not the number `0`. However, there is a notation for natural numbers which allows the user to write them using the usual decimal notation (and thus `O` as `0`, and `S (S (S O))` as `3`).

Exercises. Follow the same schema to define types for the following representations:

- the set $\mathbb{Z}$ of integers as a free structure with `zero` and two injections `pos` and `neg` from `nat` to $\mathbb{Z}$, where the term `(pos n)` stands for $n+1$ and `(neg n)` for $-n-1$;
- arithmetic expressions corresponding to the following abstract syntax:

$$\texttt{expr} ::= 0 \mid 1 \mid \texttt{expr} + \texttt{expr} \mid \texttt{expr} - \texttt{expr}$$

- binary trees over a type `A` (to be declared):

$$\texttt{tree} ::= \mathsf{empty} \mid \mathsf{node}(A, \texttt{tree}, \texttt{tree})$$

1.5.2 Inductive Predicates

Inductive definitions can be used to introduced predicates specified by a set of closure properties, as some kind of generalized Prolog clauses. Each clause is given a name, seen as a constructor of the relation and whose type is the logical formula associated to the clause.

The syntax of such a definition is:

$$\texttt{Inductive}\ \textit{name} : \textit{arity} := c_1 : C_1 \mid \ldots \mid c_n : C_n$$

where *name* is the name of the relation to be defined, *arity* its type (for instance `nat->nat->Prop` for a binary relation over natural numbers) and, as for data types, c_i and C_i the names and types of constructors respectively.

Example. The definition of the order relation over natural numbers can be defined as the smallest relation verifying:

```
forall n:nat, LE O n
forall n m:nat, LE n m -> LE (S n) (S m)
```

In Coq, such a relation is defined as follows:

```
Coq < Inductive LE : nat -> nat -> Prop :=
Coq < | LE_O : forall n:nat, LE 0 n
Coq < | LE_S : forall n m:nat, LE n m -> LE (S n) (S m).
```

This declaration introduces identifiers `LE`, `LE_O` and `LE_S`, each having the type specified in the declaration. Moreover, case analysis operations and recursive definitions are associated to inductive types which allow to capture the minimality of the relation.

Actually, the definition if this order relation in Coq standard library is slightly different:

```
Coq < Inductive le (n: nat) : nat -> Prop :=
Coq < | le_n : le n n
Coq < | le_S : forall m:nat, le n m -> le n (S m).
```

The parameter `(n:nat)` after `le` is used to factor out `n` in the whole inductive definition. As a counterpart, the first argument of `le` must be `n` everywhere in the definition. In particular, `n` could not have been a parameter in the definition of `LE` since `LE` must be applied to `(S n)` in the second clause.

Exercises.

- Define an inductive predicate `Natural` over $\mathbb{Z}$ which characterizes `zero` and the positive numbers.
- Define a relation `Diff` such that (`Diff` $n\ m\ z$) means that the value of $n-m$ is z, for two natural numbers n and m and an integer z.
- Define a relation `SubZ` which specifies the difference between two elements of $\mathbb{Z}$.
- Define a relation `Sem` which relates any expression of type `expr` to its "semantics" as an integer.

1.6 Function Definitions

1.6.1 The Pattern-Matching Operator

When a term t belongs to some inductive type, it is possible to build a new term by case analysis over the various constructors which may occur as the head of t when it is evaluated. Such definitions are known in functional programming languages as *pattern-matching*. The Coq syntax is the following:

match *term* with c_1 *args*$_1$ => *term*$_1$ ... c_n *args*$_n$ => *term*$_n$ end

In this construct, the expression *term* has an inductive type with n constructors c_1, ..., c_n. The term *term*$_i$ is the term to build when the evaluation of t produces the constructor c_i. It is possible to give the expected type for the result with the following variant:

match *term* return *type* with c_1 *args*$_1$ => *term*$_1$ ... c_n *args*$_n$ => *term*$_n$ end

Natural Numbers. If `n` has type `nat`, the function checking whether `n` is `O` can be defined as follows:

```
Coq < Check (fun n:nat => match n with
Coq <                    | O => true
Coq <                    | S x => false
Coq <                    end).
```

1.6.2 Generalized Pattern-Matching Definitions

More generally, patterns can match several terms at the same time, can be nested and can contain the universal pattern _ which filters any expression. Patterns are examined in a sequential way (as in functional programming languages) and must cover the whole domain of the inductive type. Thus one may write for instance

```
Coq < Check (fun n m:nat =>
Coq <           match n, m with
Coq <           | O, _ => true
Coq <           | _, S O => false
Coq <           | _, _ => true
Coq <           end).
```

However, the generalized pattern-matching is not considered as a primitive construct and is actually *compiled* in primitive patterns as described in the previous section.

1.6.3 Some Equivalent Definitions

In the case of an inductive type with a single constructor `C`, the construct construction :

$$\texttt{let } (x_1\texttt{,..,}x_n) \texttt{ := } t \texttt{ in } u$$

can be used as an equivalent to `match` t `with C` x_1`..`x_n `=>` u `end`.

In the case of an inductive type with two constructors $\texttt{c}_1$ and $\texttt{c}_2$ (such as the type of booleans for instance) the construct

$$\texttt{if } t \texttt{ then } u_1 \texttt{ else } u_2$$

can be used as an equivalent to `match` t `of` $\texttt{c}_1$ `_ =>` u_1 `|` $\texttt{c}_2$ `_ =>` u_2 `end`.

1.6.4 Dependant Elimination

To prove a property $P\,n$ on a natural number n, one may distinguish the case $n = \texttt{O}$ and the case $n = \texttt{S}\,p$ for another natural number p. Thus we simply need to prove $P\ \texttt{O}$ and `forall` p : `nat`, $P\ (\texttt{S}\ p)$. The term which encodes this proof is also built by pattern-matching using the `match` operator.

In that case, the predicate P over which the case analysis is built is usually difficult (or even impossible) to infer automatically. Thus it must be given explicitly by the user, using the following variant of `match`:

`match` *term* `as` x `return` *type*(x) `with` C_1 *args*$_1$ `=>` *term*$_1$ `...` C_n *args*$_n$ `=>` *term*$_n$ `end`

.

Exercises.

- Define the predecessor function of type `nat->nat` and a function of type $\mathbb{Z}$`->bool` which checks whether its argument is 0.
- Let `A:Set` and `P:A->Set` be parameters. Define the dependant sum of type `Set` with a single constructor `intro : forall x:A, P x ->sum`. Then define the two projections `pi1` of type `sum->A` and `pi2` of type `forall s:sum, P (pi1 s)`.

1.6.5 Fixpoint Definitions

To define functions over inductive data types, it is necessary to use recursion. General recursion cannot be used since it would lead to an unsound logic.

Only structural recursion is allowed. It means that a function can be defined by fixpoint if one of its formal arguments, say x, as an inductive type and if each recursive call is performed on a term which can be checked as structurally smaller than x. The basic idea is that x will usually be the main argument of a `match` construct and then recursive calls can be performed in each branch on some variables of the corresponding pattern.

The `Fixpoint` Construct. The syntax for a fixpoint definition is the following:

`Fixpoint` *name* `(`x_1`:`*type*$_1$`) ... (`x_p`:`*type*$_p$`) {struct` x_i`}:` *type*$_f$ `:=`*term*

The variable x_i following the `struct` keyword is the recursive argument. Its type *type*$_i$ must be an instance of an inductive type. The type of *name* is `forall (`x_1`:`*type*$_1$`)...(`x_p`:`*type*$_p$`),` *type*$_f$. Occurrences of *name* in *term* must be applied to at least i arguments and the ith must be recognized as structurally smaller than x_i. Note that the `struct` keyword may be omitted when $i = 1$.

Examples. The following two definitions of `plus` by recursion over the first and the second argument respectively are correct:

```
Fixpoint plus1 (n m:nat) {struct n} : nat :=
      match n with
      | O => m
      | S p => S (plus1 p m)
      end.
Fixpoint plus2 (n m:nat) {struct m} : nat :=
      match m with
      | O => n
      | S p => S (plus2 n p)
      end.
```

1.6.6 Combinators

In Coq, the constructs `match` and `fix` (the internal construct for `Fixpoint` definitions) are used for all definitions over inductive types. We have seen that a higher-level construct can be used for complex pattern-matching and recursive function definitions.

In other logical frameworks such as system T or Martin-Löf's type theory, we rather use combinators implementing primitive recursion or axioms corresponding to structural induction schemas.

In Coq, such combinators are automatically introduced when an inductive type is declared. For instance, the two constants Ind_`ind` and Ind_`rec` are automatically defined when the inductive type Ind is declared. Other schemas can be manually generated using command `Scheme`.

1.6.7 Computing

One can reduce a term and prints its normal form with `Eval compute in` *term*. For instance:

```
Coq < Eval compute in (fun x:nat => 2 + x) 3.
     = 5
     : nat
```

1.6.8 Exercises

- Define an `or` function over booleans. Check the properties `or true _ = true` and `or false b = b`.
- Define a function from $\mathbb{Z}$ to $\mathbb{Z}$ for the negation of an integer.
- Define the canonical injection from `nat` to $\mathbb{Z}$.
- Define a function of type `nat` $\rightarrow$ `nat` $\rightarrow \mathbb{Z}$ which computes the difference between two natural numbers, with the following specification:

$$\begin{array}{ll} \texttt{diff}\,0\,0 = \texttt{zero} & \texttt{diff}\,0\,(S\,n) = \texttt{neg}\,n \\ \texttt{diff}\,(S\,n)\,0 = \texttt{pos}\,n & \texttt{diff}\,(S\,n)\,(S\,m) = \texttt{diff}\,n\,m \end{array}$$

- Use the function `diff` above to define addition and subtraction over type $\mathbb{Z}$ (*i.e.* as functions of type $\mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$).
- Define a function which maps each expression of type `expr` to its "semantics" as an element of $\mathbb{Z}$.

1.7 Proofs

To prove a theorem with Coq, one has to state it first, with one of the following two declarations:

$$\{\texttt{Theorem} \mid \texttt{Lemma}\}\ \textit{name} : \textit{type}$$

where *name* is the name of the theorem and *type* its statement.

1.7.1 First-Order Reasoning

Coq logic uses natural deduction rules for a higher-order predicate calculus. Each connective is associated to introduction and elimination rules, as usual. For instance, the introduction rules for conjunction is

$$\frac{A \quad B}{A \wedge B}$$

whereas the two elimination rules are usually given as

$$\frac{A \wedge B}{A} \qquad \frac{A \wedge B}{B}$$

which is equivalent to the following rule used by Coq:

$$\frac{A \wedge B \qquad A \Rightarrow B \Rightarrow C}{C}$$

A tactic can be associated to each inference rule of the logic, in a natural way: starting with a goal which is an instance of the rule conclusion, we generate one subgoal for each premise of the rule, side-conditions being checked if any. For instance, conjunction is associated to the `split` tactic, which corresponds to the introduction rule. Thus it transforms a goal $A \wedge B$ into two goals A and B. For elimination rules, the information in the conclusion is not sufficient to instantiate the premises (one has to know A and B). Thus tactics may have arguments to indicate the missing information. In the case of the conjunction for instance, the tactic will take a proof of the main premise, that is a proof of $A \wedge B$. The following table gives the correspondence between Coq syntax, usual connectives and introduction and elimination tactics.

Proposition (P)	Coq syntax	Introduction	Elimination (H of type P)
$\perp$	`False`		`elim` H, `contradiction`
$\neg A$	`~A`	`intro`	`apply` H
$A \wedge B$	`A/\B`	`split`	`elim` H
$A \Rightarrow B$	`A->B`	`intro`	`apply` H
$A \vee B$	`A\/B`	`left, right`	`elim` H
$\forall x : A.P$	`forall (x:A), P`	`intro`	`apply` H
$\exists x : A.P$	`exists (x:A), P`	`exists` *witness*	`elim` H
$x =_A y$	`x=y`	`reflexivity, trivial`	`elim` H, `rewrite` H

The `assumption` Tactic. The axiom rule is implemented by the `assumption` tactic, which searches the context for a proof of the current goal.

The `apply with` and `elim with` Tactics. In elimination rules, the argument of the `apply` and `elim` tactics is a term which proves the main premise. Usually, one does not have a theorem or an hypothesis H which exactly proves the main premise P but a generalization, typically `forall` $(x_1 : A_1)..(x_n : A_n)$`,` P'. The tactics try to find out an adequate instance of H which can be eliminated and will generate additional subgoals if necessary. If this instance cannot be inferred automatically, the `apply` and `elim` tactics fail. Then some variants can be used to explicitly provide the terms which cannot be inferred: $\{$`apply | elim`$\}$ H `with` $t_1 \ldots t_k$.

Some Useful Commands. The following commands can be used to build a theorem proof:

- `Proof` *term*: gives an explicit proof term for the current theorem.
- `Qed`: saves a proof in the environment once it is completed (no more subgoal).
- `Admitted` : aborts a proof and replaces the statement by an axiom that can be used in later proofs.

Exercises. Prove the following tautologies:

$$\begin{array}{r|c} A \wedge (B \vee C) \Rightarrow (A \wedge B) \vee (A \wedge C) & \neg\neg\neg A \Rightarrow \neg A \\ A \vee (\forall x.(P\ x)) \Rightarrow \forall x.(A \vee (P\ x)) & \exists x.\forall y.(Q\ x\ y) \Rightarrow \forall y.\exists x.(Q\ x\ y) \end{array}$$

1.7.2 Combining Tactics

The basic tactics can be combined into more powerful tactics using tactics combinators, also called *tacticals*. Here are some of them:

Tactical	**Meaning**
t_1 `;` t_2	applies tactic t_1 to the current goal and then t_2 to each generated subgoal
t_1 `||` t_2	applies tactic t_1; if it fails then applies t_2
t `; [` t_1 `| ...|` t_n`]`	applies t and then t_i to the i-th generated subgoals; there must be exactly n subgoals generated by t
`idtac`	does nothing
`try` t	applies t if it does not fail; otherwise does nothing
`repeat` t	repeats t as long as it does not fail
`solve` t	applies t only if it solves the current goal

1.7.3 Equational Reasoning

Proving Equalities. If two terms t and u are convertible, then the `reflexivity` tactic solves the goal $t = u$. The `symmetry` tactic replaces the goal $t = u$ by the goal $u = t$. To use a transitivity step, one uses the `transitivity` v tactic which produces two subgoals $t = v$ and $v = u$.

Using Equality to Rewrite. The elimination rule for equality says that if there is a proof of $t = u$ then for each property P such that $P(t)$ holds, we also have $P(u)$. The elimination tactic on a proof of $t = u$ will replace the goal $P(t)$ by the goal $P(u)$. In practice, the effect is to replace each occurrence of u by t in the goal.

Some variants are:

- If `H` proves $t = u$ then one can replace t by u using the tactic `rewrite -> H`.
- To replace t by u while generating the subgoal $u = t$ one uses the tactic `replace` t `with` u.
- To replace some (but not all) occurrences of u, one first uses the tactic `pattern` u `at` *occs*, where *occs* is a list of occurrences to be rewritten, before using the `elim`, `rewrite` or `replace` tactic.

Convertibility. The tactics `simpl` and `unfold` *name* reduce the goal by transforming (some part of) the goal into a convertible term.

Exercises.

- Prove the stability of the constructors of type $\mathbb{Z}$ for equality ($n = m \Rightarrow \texttt{neg}\ n = \texttt{neg}\ m$, etc.).
- Prove the opposite direction, for instance $\texttt{neg}\ n = \texttt{neg}\ m \Rightarrow n = m$. Hint: start by defining a projection from $\mathbb{Z}$ to `nat`, or use the `injection` tactic.
- Prove one of the definitional equalities for function `diff`; for instance

$$\texttt{diff}\ (S\ n)\ (S\ m) = \texttt{diff}\ n\ m$$

1.7.4 Proofs by Induction

The `induction` **Tactic.** The tactic to perform proofs by induction is `induction` *`term`* where *term* is an expression in an inductive type. It can be an induction over a natural number or a list but also a usual elimination rule for a logical connective or a minimality principle over some inductive relation. More precisely, an induction is the application of one of the principles which are automatically generated when the inductive type is declared.

The `induction` tactic can also be applied to variables or hypotheses from the context. To refer to some unnamed hypothesis from the conclusion (*i.e.* the left hand-side of an implication), one has to use `induction` *num* where *num* is the *num*-th unnamed hypothesis in the conclusion.

The `induction` tactic generalizes the dependent hypotheses of the expression on which induction applies. To avoid this behavior, on may clear some hypotheses with `clear`, or use `elim` or `simple induction` which are simplified versions of `induction`.

Induction over Data Types. If *term* has an inductive type then the induction used in the natural generalization of the induction over natural numbers. The main difficulty is to tell the system what is the property to be proved by induction. The default (inferred) property is the abstraction of the goal w.r.t. all occurrences of *term*. If only some occurrences must be abstracted (but not all) then the `pattern` tactic must be applied first (as we did above for the `rewrite` tactic).

It is sometimes necessary to generalize the goal before doing the induction. This can be done using the `cut` *`type`* tactic, which changes the goal G into *type* $\Rightarrow G$ and generates a new subgoal *type*. If the generalization must include some hypotheses, one may use the `generalize` tactic first (if `x` is a variable of type A, then `generalize x` changes the goal G into the new goal `forall x:`A`,` G.

Induction over Proofs. If *term* belongs to an inductive relation then the elimination tactic corresponds to the use of the minimality principle for this relation. Generally speaking, the property to be proved is $(I\ x_1 \dots x_n) \Rightarrow G$ where I is the inductive relation. The goal G is abstracted w.r.t. $x_1 \dots x_n$ to build the relation used in the induction. It works automatically when $x_1 \dots x_n$ are variables. If not, the system cannot infer a well-typed abstraction or infers a non-provable property. In that case, the tactic `inversion` *`term`* is to be preferred (it exploits all information in $x_1 \dots x_n$).

Exercises.

- Prove the properties $\texttt{Diff}\ n\ n\ \texttt{zero}$ and $\texttt{Diff}\ (S\ n)\ n\ (\texttt{pos}\ 0)$.
- Prove that for all natural numbers n and m, if $m \leq n$ then there exists z such that $\texttt{Natural}\ z$ and $\texttt{Diff}\ n\ m\ z$.
- Prove that for all natural numbers n and m the property $\texttt{Diff}\ n\ m\ (\texttt{diff}\ n\ m)$ holds.

2 Programming with Coq

We propose a few exercises as an introduction to certified functional programming using Coq.

2.1 Functional Programming with Coq

We assume an environment where a parameter type `A` is given, as well as an order relation `inf` over `A`, of type `A` → `A` → `bool`.

1. **[Type `list`, primitive recursive functions `singl,lgth,app,rev`]** Define the data type `list` for the lists of elements of type `A`. Define a function `singl` which takes an element and builds a list containing only this element, a function `lgth` computing the length of a list, a function `app` catenating two lists and a function `rev` to reverse a list (either using `app` or using an accumulator to get a recursive terminal function).

2. **[Function Evaluation]** Introduce some variables `a,b,c` of type `A`. Test the functions above on small examples using the `Eval compute` command.

3. **[Function `nb_occ`]** Using `inf`, define a function checking for the equality of two terms of type `A`. Then define a function `nb_occ` computing the number of occurrences of a given element in a given list.

4. **[Insertion in a sorted list]** Assuming a given element and a sorted list, define a function inserting the element in the list such that the resulting list is sorted.

2.2 Quicksort

1. **[Function `split_list`]** Define a function `split_list` which takes a term `a` of type `A` and a list `l` and returns two lists, one containing the elements of `l` smaller than `a` and the other containing the elements of `l` greater or equal than `a`. Show that both lists have a length smaller or equal than the length of `l`.

2. **[Well-founded induction]** Define a function which takes a list as argument and returns a sorted list containing the same elements. It will use a divide-and-conquer approach (quicksort), as follows: is the list is not empty, we take its first element, split the remaining list according to this element, sort the two sublists recursively and finally catenate the three parts (the two sorted sublists and the element) to get a sorted list. Hint: use the well-founded induction operator `well_founded_induction` from the standard library (module `Wf_nat`).

2.3 Specifying and Proving Programs

1. **[Predicate `equiv`]** Define a predicate `equiv` of type `list`→`list`→`Prop` which holds when any element occurs exactly the same number of times in each list.

2. Prove that the insertion of `a` in the sorted list `l` is a list equivalent (that is, `equiv`) to `(cons a l)`.

3. **[Predicate `all`]** Given an arbitrary property `R:A`→`Prop`, define a property `all` which says that all the elements in a given list satisfy `R`.

4. Show that if `(all l)` and `(R a)` then `(all (insert a l))`.

5. **[Elimination over booleans]** The induction principle over booleans says that to show `(P b)` it is sufficient to prove `(P true)` and `(P false)`. Prove a stronger principle which says that to prove `(P b)` it is sufficient to prove `b=true→(P true)` and `b=false→(P false)`.

6. **[Predicate `sorted`]** Define a predicate `sorted` to characterize sorted lists. Show that the insertion into a sorted list is also a sorted list.

2.4 Programming with Proofs

1. Show the following property by induction over `l`:

   ```
   forall (a:A)(l:list), sorted l ->{m:list | equiv m (cons a l) & sorted m}
   ```

2. Is it possible to perform an induction over hypothesis `(sorted l)`?

3. Show an induction principle `sorted_rec` similar to `sorted_ind` but where predicate `P` has type `list→Set`. What is now the answer to the previous question?

4. Give function `split_list` a specification and prove its correctness.

5. Prove the following theorem: `forall l:list, {m:list | sorted m & equiv l m}`

3 Practical Use of Coq

3.1 Compilation

It is advisable to split large developments into several files. Then each file can be compiled (with the Coq compiler `coqc`) in order to be subsequently loaded in a very efficient way. Each file `f.v` is compiled in a file `f.vo` to be loaded with the `Require` command (as seen above).

A Unix tool `coq_makefile` is provided to generate a `Makefile` to automate such a compilation. It is used as follows:

```
unix% coq_makefile f1.v ... fn.v -o Makefile
```

The dependencies between the various Coq files are maintained into the file `.depend` (initially, it has to be created by the user, using for instance the command `touch .depend`). Then it can be updated with the command:

```
unix% make depend
```

3.2 Notations

3.2.1 Implicit Arguments

Some typing informations in terms are redundant. For instance, let us consider the constructor of polymorphic pairs:

```
Coq < Check pair.
pair
     : forall A B : Type, A -> B -> A * B
```

To build a pair of two natural numbers, it is not necessary to give the four arguments, but only the last two, since types `A` and `B` can be inferred as the types of the last two arguments, respectively:

```
Coq < Check (pair 0 0).
(0, 0)
     : nat * nat
```

A general mechanism, called *implicit arguments*, allows such shortcuts. It defines a set of arguments that can be inferred from other arguments. More precisely, if the type of a constant c is `forall` $(x_1 : \mathit{type}_1) \ldots (x_n : \mathit{type}_n), t$ then argument x_i is considered implicit if x_i is a free variable in one of the types type_j, in a position which cannot be erased by reduction. Such arguments are then omitted. This mechanism is enabled with the following command:

```
Coq < Set Implicit Arguments.
```

Then one can define for instance:

```
Coq < Definition pair3 (A B C:Set) (x:A) (y:B) (z:C) :
Coq <   A * (B * C) := pair x (pair y z).
```

and implicit arguments can be inspected using the `Print Implicit` command:

```
Coq < Print Implicit pair3.
pair3 : forall A B C : Set, A -> B -> C -> A * (B * C)
Arguments A, B, C are implicit
```

If the constant is applied to an argument then this argument is considered as the first non implicit argument. A special syntax `@pair3` allows to refer to the constant without implicit arguments. It is also possible to specify an explicit value for an implicit argument with syntax `(x:=t)`. Here are some examples:

```
Coq < Check (pair3 0 true 1).
pair3 0 true 1
     : nat * (bool * nat)

Coq < Check (pair3 (A:=nat)).
pair3 (A:=nat)
     : forall B C : Set, nat -> B -> C -> nat * (B * C)

Coq < Check (pair3 (B:=bool) (C:=nat) 0).
pair3 (B:=bool) (C:=nat) 0
     : bool -> nat -> nat * (bool * nat)
```

The generation of implicit arguments can be disabled with the command

```
Coq < Unset Implicit Arguments.
```

Finally, it is also possible to enforce some implicit arguments. For instance, it is possible to keep only `A` as an implicit argument for `pair3`, as follows:

```
Coq < Implicit Arguments pair3 [A].
Coq < Print Implicit pair3.
pair3 : forall A B C : Set, A -> B -> C -> A * (B * C)
Argument A is implicit
```

Since version 8.0 of Coq, all constants in the standard library are defined with the implicit arguments mechanism enabled.

3.2.2 Incomplete Terms

A subterm can be replaced by the symbol _ if it can be inferred from the other parts of the term during typing.

```
Coq < Parameter f : forall (n:nat), n=O -> nat.
Coq < Check (f _ (refl_equal O)).
```

(Note that with implicit arguments enabled, the first argument of `f` could be simply omitted.)

3.2.3 Untyped Abstractions

If the expected type for a variable can be inferred during typing, then it can be omitted in binders:

```
Coq < Definition istrue b := b = true.
Coq < Definition istrue' := fun b => b = true.
```