LASER 2011 Summerschool
Elba Island, Italy
Basics of COQ

Christine Paulin-Mohring

September 2, 2011

Contents

1 First steps in COQ

1.1 Launching COQ

The COQ system is available at `http://coq.inria.fr`. This document has been prepared using COQ version `V8.3pl2`.

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 when building a proof. The right-bottom window contains the output of commands.

A COQ *command ends with a dot.*

The COQ reference manual is available online at `http://coq.inria.fr/refman/` and accessible from the `Help` menu of `coqide`. It provides details for the different commands presented in this note.

We also recommend the course notes *Coq in a Hurry* by Y. Bertot available at `http://cel.archives-ouvertes.fr/inria-00001173`, as an alternative quick introduction to the COQ system.

More advanced books to learn COQ includes the book by Y. Bertot and P. Casteran known as the COQ'Art [1]. The course by B. Pierce on software foundations [6] using COQ is available online. The book by A. Chlipala [4] concentrates on programming with COQ and make intensive use of dependent types.

1.2 Syntax of Terms

COQ objects represent types, propositions, terms and proofs. Every object as a type (which is itself a COQ object). In COQ, $t : T$ represents the fact that t is an object of type T.

Types. Useful types are:

`nat`	natural numbers (`3:nat`)
`bool`	boolean values (`true:bool`, `false:bool`)
`Prop`	type of logical properties (`False:Prop`)
`Type` (or `Set`)	type of types (`nat:Set`, `Prop:Type`)
`T1 -> T2`	type of functions from `T1` to `T2`
`T -> Prop`	type of unary predicate on `T`
`T1 * T2`	type of pairs of objects of type `T1` and `T2`

Propositional connectives. This is a summary of COQ syntax for logical propositions (first line presents paper notation and second line the corresponding COQ input).

$\perp$	$\top$	$t = u$	$t \neq u$	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
`False`	`True`	`t=u`	`t<>u`	`~P`	`P /\ Q`	`P \/ Q`	`P -> Q`	`P <-> Q`

The arrow associates to the right such that `T1->T2->T3` is interpreted as `T1->(T2->T3)`

Quantifiers. Syntax for universal and existential quantifiers is given below with possible variants:

$\forall x, P$	`forall x, P`	`forall x:T, P`	`forall (x y:T) (z:U),P`
$\exists x, P$	`exists x, P`	`exists x:T, P`	*no multiple bindings*

Remark on Coq theory. Universal quantification is an essential construction in Coq which serves different purposes. It can be used for first-order quantification like in $\forall x :, x = x$, but also for higher-order quantification like in $\forall A : \mathsf{Type}, \forall x : A, x = x$ or $\forall A : \mathsf{Prop}, A \Rightarrow A$, both the functional arrow and the logical implication are special cases of universal quantification with no dependencies: $T_1 \to T_2$ stands for $\forall \_ : T_1, T_2$.

Syntax of functional terms.

- *Application* of function f to term t is written $f\ t$ or with parentheses $(f\ t)$. It associates to the left such that $f\ t_1\ t_2$ represents $(f\ t_1)\ t_2$.
- *Abstraction* of term t with respect to variable x is written **fun** $x \Rightarrow t$ or **fun** $(x : T) \Rightarrow t$ to indicate the type T of variable x.
- *A local definition* is introduced by **let** x :=t_1 **in** t_2.
- *Pairs* are written (t_1, t_2) and have type $T_1 * T_2$ when t_i has type T_i, the components of a pair can be accessed using the notation: **let** (x, y) :=t_1 **in** t_2.

In the following, *term* will denote any Coq term, *name* or *id* represents an identifier, *type* represents a so-called “type” which is a term with type Type, Set or Prop. We use *prop* instead of *type* when we expect a term of type *prop*, however the same commands will usually also work with a more general type.

1.3 Queries in Coq

The following interactive commands are useful to find information in libraries when doing proofs. They can be executed from the `coqide` so called *Command Pane* (use the menu `Queries` or `Windows/Show Query Pane` and then click on new page on the left of the panel).

- `Check` *term*: checks if *term* can be typed and displays its type.

```
Coq < Check (2,true).
(2, true)
     : nat * bool

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

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

Application: What is proved by lemma `bool_ind`? Are the following terms well-typed: `0+1`, `true+false`?

- `SearchAbout` *name*: displays all declarations *id* : *type* in the environment such that *name* appears in *type*.

```
Coq < SearchAbout fst.
surjective_pairing: forall (A B : Type) (p : A * B), p = (fst p, snd p)
injective_projections:
  forall (A B : Type) (p1 p2 : A * B),
  fst p1 = fst p2 -> snd p1 = snd p2 -> p1 = p2
```

  Application: Find all lemmas about conjunction `and`, disjunction `or`, equality `eq` and order relation `le`.
  Useful variants are `SearchAbout` [$name_1 \cdots name_n$] to find objects with types mentioning all the names $name_i$ and also `SearchAbout` *pattern* to find objects with types mentioning an instance of the *pattern* which is a term possibly using the special symbol "_" to represent an arbitrary term.

```
Coq < SearchAbout [plus 0].
plus_n_O: forall n : nat, n = n + 0
plus_O_n: forall n : nat, 0 + n = n

Coq < SearchAbout ( ~ _  <-> _ ).
neg_false: forall A : Prop, ~ A <-> (A <-> False)
```

- `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`. Find the definitions of operations `plus` and `mult`. Note that the latter are printed as infix symbols + and $\star$.

1.4 Loading new libraries

The command `Require Import` *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 commands `Print Libraries` 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 gathered in a single module `Arith` in such a way that the command `Require Import Arith` loads and opens all these modules.

```
Coq < Require Import Arith.
```

Applications: Display the loadpath. Load the module `Bool` on boolean operations and search for theorems about the boolean conjunction `andb`.

The command `Require` *name* only loads the library, the objects inside are refered by a qualified name: *name.id*. This long name is also useful when the same identifier exists in different libraries. The command `Locate` *id* helps find all occurrence of *id* in loaded libraries.

1.5 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 with the `Require` command.

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
```

2 Doing basic proofs with COQ

To prove a theorem, one first enunciates the corresponding statement using declaration:

`Theorem` *name* : *prop* or `Lemma` *name* : *prop* or `Goal` *prop*

where *name* is the name of the theorem (for later reference) and *prop* the type corresponding to its statement.

A proof in COQ is developed interactively using *tactics*. A tactic is a program which transforms a goal to be proved into a (possible empty) set of new subgoals which are sufficient conditions to establish the original result.

The `Theorem` `Lemma` or `Goal` instructions generate a top-level goal. After each tactic application, the system will display all the goals which remain to be proved in order to finish the proof of the theorem. At the end, when there is no more subgoals, the command `Qed` will construct from the tactics a proof-term that will be saved as the definition of the theorem for further reuse.

In `coqide` you can use the arrows up and down in the menu to navigate in your script interpreting tactics and undoing them. The part which as been interpreted is highlighted in green and cannot be edited in order to preserve the system consistency.

If the proof is not finished then the command `Admitted` can be used and will introduce the theorem as an axiom.

The tactic `admit` will just admit the current subgoal as an axiom and skip to the following subgoals to be solved.

2.1 First-Order Reasoning

Tactic for explicit proof. The basic case to solve a goal corresponding to a proposition P is to produce a term t of type P (usually an hypothesis). The tactic `exact` t will solve the goal.

The `assumption` tactic searches in the hypotheses of the goal an exact proof for the current conclusion.

Tactic for logical connectives. COQ logic uses natural deduction rules for a higher-order (intuitionistic) 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, the introduction rule for conjunction is implemented by the `split` tactic. 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 tactics implementing introduction and elimination rules.

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

It is highly recommended to give explicit names to objects introduced during the proof but the tactics `intro` and `destruct` can also be used without explicit naming.

```
Coq < Lemma conj_sym : forall A B, A /\ B -> B /\ A.
1 subgoal

  ============================
   forall A B : Prop, A /\ B -> B /\ A

Coq < intros A B p.
1 subgoal

  A : Prop
  B : Prop
  p : A /\ B
  ============================
   B /\ A

Coq < destruct p as (a,b).
1 subgoal

  A : Prop
  B : Prop
  a : A
  b : B
  ============================
   B /\ A

Coq < split.
2 subgoals
```

```
  A : Prop
  B : Prop
  a : A
  b : B
  ============================
   B
subgoal 2 is:
 A

Coq < exact b. (* or assumption *)
1 subgoal

  A : Prop
  B : Prop
  a : A
  b : B
  ============================
   A

Coq < exact a. (* or assumption *)
Proof completed.

Coq < Qed.
intros A B p.
destruct p as (a, b).
split.
 exact b.

 exact a.

conj_sym is defined
```

Exercises. Prove the following tautologies:

$A \wedge (B \vee C) \Rightarrow (A \wedge B) \vee (A \wedge C)$	$\neg\neg\neg A \Rightarrow \neg A$
$A \vee (\forall x.(P\ x)) \Rightarrow \forall x.(A \vee (P\ x))$	$\exists x.\forall y.(Q\ x\ y) \Rightarrow \forall y.\exists x.(Q\ x\ y)$

The fact that Coq logic is intuitionistic implies that there are no way to prove $A \vee \neg A$ or $\neg\neg A \Rightarrow A$ of $\forall x, P \vee \exists x, \neg P$ for arbitrary properties A and P. The quantifiers $\exists$ and $\vee$ have a stronger meaning than in classical logic: they are interpreted as the existence of an effective way to compute the witness for an existential or the case for a disjunction. A lot of property does not require the use of classical logic. But if needed, Coq provides a library which adds the axiom of excluded middle and derive useful consequences like.

```
Coq < Require Import Classical.

Coq < Check not_all_not_ex.
not_all_not_ex
     : forall (U : Type) (P : U -> Prop),
       ~ (forall n : U, ~ P n) -> exists n : U, P n
```

It is possible to test if a given theorem depends on unproved assumptions:

```
Coq < Print Assumptions not_all_not_ex.
Axioms:
classic : forall P : Prop, P \/ ~ P
```

2.2 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
`try` t	applies t if it does not fail; otherwise does nothing
`repeat` t	repeats t as long as it does not fail

2.3 Equational Reasoning

Proving Equalities. We give a correspondence between standard rule for equality and Coq tactics. The relation $t \equiv u$ means that t and u represents the same value after computation (we say t and u are convertible).

`reflexivity`	$\dfrac{t \equiv u}{t = u}$
`symmetry`	$\dfrac{u = t}{t = u}$
`transitivity` v	$\dfrac{t = v \quad v = u}{t = u}$
`f_equal`	$\dfrac{f = g \quad t_1 = u_1 \dots t_n = u_n}{f\,t_1 \dots t_n = g\,u_1 \dots u_n}$

Using Equality to Rewrite. The elimination rule for equality is :

$$\frac{t = u \qquad P(t)}{P(u)}$$

it is implemented by the tactic `replace` u `with` t.

Very often one knows a proof H of $t = u$ (or a generalisation of it) and one can use the tactic `rewrite <- H` or simply `rewrite H` when H proves $u = t$ to perform the rewriting in the goal.

The `rewrite` tactic by default replace all the occurrences of u in $P(u)$. To rewrite selected occurrences, there is a variant: `rewrite` H `at` *occs*.

Another useful tactic for dealing with equalities is `subst`. It x is a variable and the context contains an hypothesis $x = t$ (or $x = t$) with x not occurring in t, then the tactic `subst` x will substitute t for x and remove both x and the hypothesis from the context. The tactic `subst` without argument do the substitution on all possible variables in the context.

2.4 Guiding the proof process

The `apply with` tactic. In elimination rules for implication and universal quantification, the main premise is a proof of $A \Rightarrow B$ or `forall` x, P.

Usually, one does not have a theorem or an hypothesis H which exactly proves the main premise P but a generalisation, typically `forall` $(x_1 : A_1)..(x_n : A_n), P'$ (remember that internally `forall` capture also implication). The tactic `apply` works in this more general case. It tries 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` tactics fails. Then some variants can be used to explicitly provide missing information:

- `apply` *H* `with` $t_1 \dots t_k$ where $t_1 \dots t_k$ are exactly the missing arguments.
- `apply` *H* `with` $(x_i := t_i)$ to give explicitly an argument.

A typical case where it is needed is with transitivity proofs as shown below:

```
Coq < Check le_trans.
le_trans
     : forall n m p : nat, n <= m -> m <= p -> n <= p

Coq < Goal forall x y, x <= 2 -> 2 <= y -> x <= y.
1 subgoal

  ============================
   forall x y : nat, x <= 2 -> 2 <= y -> x <= y

Coq < intros x y H1 H2.
1 subgoal

  x : nat
  y : nat
  H1 : x <= 2
  H2 : 2 <= y
  ============================
   x <= y

Coq < apply le_trans.
Toplevel input, characters 6-14:
> apply le_trans.
>       ^^^^^^^^
Error: Unable to find an instance for the variable m.
```

`apply` compares the current goal with the conclusion of `le_trans` leading to values for n and p but does not guess how to instantiate the middle value m which has to be given explicitly.

```
Coq < apply le_trans with 2.
2 subgoals

  x : nat
  y : nat
  H1 : x <= 2
  H2 : 2 <= y
  ============================
   x <= 2
subgoal 2 is:
 2 <= y
```

or alternatively:

```
Coq < apply le_trans with (m:=2).
2 subgoals

  x : nat
  y : nat
  H1 : x <= 2
  H2 : 2 <= y
  ============================
   x <= 2
subgoal 2 is:
 2 <= y
```

The `rewrite` tactic may also need partial instantiation information to work properly.

Introducing intermediate steps with `assert`. The tactic style of proof development is centred on the goal which is transformed until no more subgoals are left.

Sometimes it is useful to work in a more direct way, deducing facts from hypotheses. To help achieve this kind of reasoning, the tactic `assert` *prop* will introduce *prop* as a new goal to be proved and add *prop* as a new hypothesis of the current goal.

```
Coq < Goal forall (f : nat->nat) a b c, b = c -> f b = c -> f c = a -> c = a.
1 subgoal

  ============================
   forall (f : nat -> nat) (a b c : nat),
   b = c -> f b = c -> f c = a -> c = a

Coq < intros f a b c H1 H2 H3.
1 subgoal

  f : nat -> nat
  a : nat
  b : nat
  c : nat
  H1 : b = c
  H2 : f b = c
  H3 : f c = a
  ============================
   c = a

Coq < assert (f b = f c).
2 subgoals

  f : nat -> nat
  a : nat
  b : nat
  c : nat
  H1 : b = c
  H2 : f b = c
  H3 : f c = a
  ============================
   f b = f c
subgoal 2 is:
 c = a
```

```
Coq < apply f_equal; assumption.
1 subgoal

  f : nat -> nat
  a : nat
  b : nat
  c : nat
  H1 : b = c
  H2 : f b = c
  H3 : f c = a
  H : f b = f c
  ============================
   c = a

Coq < rewrite <- H2; rewrite <- H3; assumption.
Proof completed.
```

2.5 Automated proofs

The previous section present tactics corresponding to atomic steps of deduction. COQ has also more advanced tactics to solve complex goals. COQ is build on a safe kernel, a complex automated tactic ultimately generates a proof term which is checked again when the proof is finished.

Applying automatically known results. The `auto` tactic uses databases of known lemmas that are successively tried in order to complete the current goal. It performs introductions and conclude with assumptions. If the goal is not solved it is left unchanged. The user may add new lemmas to a database using the command `Hint Resolve` *name* and also look at the hints database applicable to current goal using command `Print Hint`.

```
Coq < Lemma pair3 : forall A B C : Prop, A -> B -> C -> A /\ B /\ C.
1 subgoal

  ============================
   forall A B C : Prop, A -> B -> C -> A /\ B /\ C

Coq < auto.
Proof completed.
```

The `auto` tactic does not try to decompose properties in the environment such that the following application of `auto` does not make any progress.

```
Coq < Lemma pair3 : forall A B C : Prop, (A /\ C) -> B -> A /\ B /\ C.
1 subgoal

  ============================
   forall A B C : Prop, A /\ C -> B -> A /\ B /\ C

Coq < auto.
1 subgoal

  ============================
   forall A B C : Prop, A /\ C -> B -> A /\ B /\ C
```

The tactic `intuition` will first destruct the propositional connectives before applying `auto` on the generated goals.

```
Coq < intuition.
Proof completed.
```

The `trivial` tactic is a variant of `auto` which only tries trivial lemmas, not generating subgoals.

Solving arithmetical problems. The `omega` tactic solves propositional problems from linear arithmetic (also known as Presburger arithmetic) involving only addition, equalities, inequality. It works on natural numbers or on integers.

```
Coq < Require Import Omega.

Coq < Lemma neq_equiv : forall x y, x <> y <-> x < y \/ y < x.
1 subgoal

  ============================
   forall x y : nat, x <> y <-> x < y \/ y < x

Coq < intros x y; omega.
Proof completed.
```

When multiplication is involved, a useful automated tactic is `ring` which solves consequences of ring properties (or semi-ring properties in the case of `nat`).

```
Coq < Lemma ring_ex : forall x y z, x * y + x * z = (z + y) * x.
1 subgoal

  ============================
   forall x y z : nat, x * y + x * z = (z + y) * x

Coq < intros; ring.
Proof completed.
```

Exercise. Prove the following property: $\forall f, (\forall xy, f\,(x+y) = fx + fy) \Rightarrow f\,0 = 0$.

3 Introducing new Coq objects

The language of Coq is not limited to basic logic and natural numbers. It is possible to introduce new objects. These can be definitions (abbreviation for complex terms) or undefined objects presented in an axiomatic way. The other way to enrich a theory is the use of a general mechanism for inductive definitions that will be introduced in the next section.

3.1 Definitions

A new definition is introduced by:

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

The identifier *name* is then an abbreviation for the term *term*. The type *type* is optional.

Example. The square function can be defined as follows:

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

or equivalently as follows:

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

A COQ definition *name* can be unfolded in a goal by using the tactic `unfold` *name* (in the conclusion) or `unfold` *name* `in` H (in hypothesis H).

3.2 Parameters and sections

The logic of COQ is powerful enough to develop inside a large part of mathematics, such that a theory will be a set of definitions and theorems and does not require to introduce axioms.

However, it is sometimes useful to be able to introduce parameters for the theory under development. The syntax is:

`Parameter` *name* : *type* or `Axiom` *name* : *prop*

where *name* is the name of the hypothesis or variable to introduce and *type* its type. The following specification introduces a type A with only one element $a : A$.

```
Coq < Parameter A : Type.
A is assumed

Coq < Parameter a : A.
a is assumed

Coq < Axiom A1 : forall y:A, y=a.
A1 is assumed
```

However, it is up to the user to make sure the axioms introduced do not lead to a contradiction.

Sections It is often convenient to introduce a local context of variables and properties, which are shared between several definitions. It is done with a section mechanism. A section *name* is opened using the command `Section` *name*. Then objects can be introduced using the syntax:

`Variable` *name* : *type* or `Hypothesis` *name* : *prop*

Several variables with the same type can be introduced with a single command, using the variants `Variables` and `Hypotheses` and a blank-separated list of names. The following definitions can refer to the objects in the context of the section. The section is ended by the command `End` *name*; then all definitions are automatically abstracted with respect to the variables they depend on.

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

```
Coq < Section test.

Coq < Variable A : Type.
A is assumed
```

```
Coq < Variables x y : A.
x is assumed
y is assumed

Coq < Definition double : A * A := (x,x).
double is defined

Coq < Definition triple : A * A * A := (x,y,x).
triple is defined

Coq < End test.
```

After ending the section, the objects A, x and y are not accessible anymore and one can observe the new types of `double` and `triple`.

```
Coq < Print double.
double =
fun (A : Type) (x : A) => (x, x)
     : forall A : Type, A -> A * A
Argument scopes are [type_scope _]

Coq < Print triple.
triple =
fun (A : Type) (x y : A) => (x, y, x)
     : forall A : Type, A -> A -> A * A * A
Argument scopes are [type_scope _ _]
```

3.3 Notations

The Coq kernel interprets a term in which all the type information is present. It makes the type-checking mechanism easier it also makes the term unreadable and writing them very cumbersome.

Coq provides different mechanisms to hide part of the term structure either for input or output; the system being responsible to build the missing information before the term is sent to the kernel.

Implicit Arguments. Some typing information in terms is 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 to be 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` $(x_1 : \mathit{type}_1) \ldots (x_n : \mathit{type}_n)$, *type* 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) :  A * (B * C)
Coq <    := pair x (pair y z).
pair3 is defined
```

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
```

Incomplete Terms A subterm can be replaced by the symbol _ if it can be inferred from the other parts of the term during type-checking.

```
Coq < Check (pair3 _ _ 0 1 2).
pair3 nat nat 0 1 2
     : nat * (nat * nat)
```

More on hiding information Other powerful techniques allow to infer automatically part of the term. For instance the coercion mechanism allows to use an object of type A when an object of type B is expected by silently applying a user-declared function (coercion) from A to B. For instance a boolean value b : `bool` can be considered as an object of type Prop using the function **fun** $b \Rightarrow b =$ `true`.

The `Display` menu allows to turn off some of the pretty-printing options; it can sometimes be useful for debugging proofs.

4 Inductive Declarations

Inductive definitions are another main ingredient of Coq language. It is a generic mechanism which captures different notions such as data-types, logical connectives, primitive relations.

4.1 Inductive Data Types

A data-type *name* can be declared by specifying a set of constructors. Each constructor c_i is given a type C_i which declare the type of its expected arguments. A constructor possibly accepts arguments (which can be recursively of type *name*), and when applied to all its arguments, a constructor has type the inductive definition *name* itself. There are some syntactic restrictions over the type of constructors to make sure that the definition is well-founded.

The syntax for declaring an inductively defined type is:

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

where *name* is the name of the type to be defined; *sort* is one of Set or Type; c_i are the names of the constructors and C_i is the type of constructor c_i.

The declaration of an inductive definition introduces new primitive objects for the type itself and its constructors it also generates theorems which are abbreviations for terms combining pattern-matching and possibly a fixpoint which proves induction principles.

Examples. The data type of booleans and natural numbers are defined inductively as follows:

```
Coq < Print bool.
Inductive bool : Set :=  true : bool | false : bool

Coq < Check bool_ind.
bool_ind
     : forall P : bool -> Prop,
       P true -> P false -> forall b : bool, P b

Coq < Print nat.
Inductive nat : Set :=  O : nat | S : nat -> nat
For S: Argument scope is [nat_scope]

Coq < Check nat_ind.
```

```
nat_ind
     : forall P : nat -> Prop,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

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 scheme 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:

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

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

$$\texttt{list} ::= \mathsf{nil} \mid \mathsf{cons}(A, \texttt{list})$$

Inductive type and equality The constructors of an inductive type are injective and distinct. For instance for natural numbers, one can prove $\texttt{S}\,n = \texttt{S}\,m \to n = m$ and $\texttt{S}\,n \neq 0$. These lemmas are part of the standard library for natural numbers but have to be proved for new inductive types. There are tactics to automate this process.

- `discriminate` H will prove any goal if H is a proof of $t_1 = t_2$ with t_1 and t_2 starting with different constructors. With no argument `discriminate` will try to find such a contradiction in the context.
- `injection` H assumes H is a proof of $t_1 = t_2$ with t_1 and t_2 starting with the same constructor. It will deduce equalities $u_1 = u_2, v_1 = v_2, \ldots$ between corresponding subterms and add these equalities as new hypotheses.

```
Coq < Goal (forall n, S (S n) = 1 -> 0=1).
1 subgoal

  ============================
   forall n : nat, S (S n) = 1 -> 0 = 1

Coq < intros n H.
1 subgoal

  n : nat
  H : S (S n) = 1
  ============================
   0 = 1

Coq < discriminate H.
Proof completed.
```

```
Coq < Goal (forall n m, S n = S (S m) -> 0 < n).
1 subgoal

  ============================
   forall n m : nat, S n = S (S m) -> 0 < n

Coq < intros n m H.
1 subgoal

  n : nat
  m : nat
  H : S n = S (S m)
  ============================
   0 < n

Coq < injection H.
1 subgoal

  n : nat
  m : nat
  H : S n = S (S m)
  ============================
   n = S m -> 0 < n
```

Remark on inductive propositions. (only if you want to better understand Coq underlying theory)

The *sort* in an inductive definition can also be Prop allowing the inductive declaration of logical propositions. Following the Curry-Howard correspondence between proposition and types, all propositional connectives except for negation, implication and universal quantifier are declared using inductive definitions. `False` is a degenerate case where there are no constructors. `True` is the proposition with only one proof `I` corresponding to the `unit` type with only one constructor.

```
Coq < Print False.
Inductive False : Prop :=

Coq < Check False_ind.
False_ind
     : forall P : Prop, False -> P

Coq < Print True.
Inductive True : Prop :=  I : True
```

Exercises on inductive propositions.

- Check for the definition of conjunction (`and`) and disjunction (`or`) as well as existential quantification (`ex`).
- Try to introduce your own connector `ifp` such that `ifp` A B C is equivalent to $(A \wedge B) \vee (\neg A \wedge C)$ but defined directly as an inductive proposition with two constructors without using conjunction or disjunction.
- Prove that $\forall A\, B\, C$, `ifp` $A\ B\ C \Leftrightarrow (A \wedge B) \vee (\neg A \wedge C)$. The tactics `destruct`/`left`/`right` will also work for `ifp`.

4.2 Definitions by pattern-matching and fixpoint

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:

$$\textbf{match}\ \textit{term}\ \ \textbf{with}\ c_1\ \textit{args}_1 \Rightarrow \textit{term}_1\ \ldots\ c_n\ \textit{args}_n \Rightarrow \textit{term}_n\ \textbf{end}$$

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:

$$\textbf{match}\ \textit{term}\ \textbf{return}\ \textit{type}\ \textbf{with}\ c_1\ \textit{args}_1 \Rightarrow \textit{term}_1\ \ldots\ c_n\ \textit{args}_n \Rightarrow \textit{term}_n\ \textbf{end}$$

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

```
Coq < Definition iszero n := match n with
Coq <                        | O => true
Coq <                        | S x => false
Coq <                        end.
iszero is defined
```

Generalised 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 < Definition nozero n m :=  match n, m with
Coq <            | O, _ => false   | _, O => false   | _, _ => true
Coq <            end.
nozero is defined
```

However, the generalised pattern-matching is not considered as a primitive construct and is actually *compiled* into a sequence of primitive patterns.

Some Equivalent Notations In the case of an inductive type with a single constructor `C`:

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

can be used as an equivalent to **match** t **with** $Cx_1..x_n \Rightarrow u$ **end**.

In the case of an inductive type with two constructors (like booleans) `c`$_1$ and `c`$_2$ (such as the type of booleans for instance) the construct

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

can be used as an equivalent to **match** t **with** $c_1 \Rightarrow u_1 | c_2 \Rightarrow u_2$ **end**.

Exercise.

- Define the predecessor function of type $\mathbb{Z} \rightarrow \mathbb{Z}$.

Fixpoint Definitions To define interesting functions over recursive data types, we use recursive functions. General fixpoints are not allowed since they 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:

$$\texttt{Fixpoint}\ \textit{name}\ (x_1 : \textit{type}_1)\ \ldots\ (x_p : \textit{type}_p)\{\texttt{struct}\ x_\texttt{i}\} : \textit{type}_f\ := \textit{term}$$

The variable x_i following the `struct` keyword is the recursive argument. Its type *type*$_i$ must be an instance of an inductive type. If the clause $\{$struct $x_i\}$ is omitted, the system will try to infer an appropriate argument.

The type of *name* is `forall` $(x_1 : \texttt{type}_1)\ldots(x_p : \texttt{type}_p),\ \texttt{type}_f$. Occurrences of *name* in *term* must be applied to at least i arguments and the ith must be recognised 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:

```
Coq < Fixpoint plus1 (n m:nat) : nat :=
Coq <       match n with
Coq <       | O => m
Coq <       | S p => S (plus1 p m)
Coq <       end.
plus1 is recursively defined (decreasing on 1st argument)

Coq < Fixpoint plus2 (n m:nat) : nat :=
Coq <       match m with
Coq <       | O => n
Coq <       | S p => S (plus2 n p)
Coq <       end.
plus2 is recursively defined (decreasing on 2nd argument)
```

A fixpoint can be computed when the recursive argument starts with a constructor. So `plus1` $0\ n$ and n are convertible but `plus1` $n\ 0$ is in normal form when n is a variable. The equation corresponding to the fixpoint definition is not trivial but can be proved by simple case analysis over the recursive argument.

```
Coq < Lemma plus1_eq : forall n m,
Coq <   plus1 n m =  match n with
Coq <       | O => m
Coq <       | S p => S (plus1 p m)
Coq <       end.
1 subgoal

  ============================
   forall n m : nat,
   plus1 n m = match n with
               | 0 => m
               | S p => S (plus1 p m)
```

```
              end
Coq < destruct n; trivial.
Proof completed.
```

The tactic `simpl` *name* when *name* is a fixpoint definition will simplify the expression whenever it is applied to a constructor. The tactic `simpl` simplifies all fixpoint definitions in the goal (which is sometimes too much, in which case it is recommended to prove the relevant equations as theorems and use them in a controled way with the `rewrite` tactic).

Remark. COQ does not prevent to define empty inductive data-types. For instance:

```
Coq < Inductive E : Set := Ei  : E -> E.
E is defined
E_rect is defined
E_ind is defined
E_rec is defined
```

But of course, there are no way to build a value (term without variable) in type E and furthermore, one can build a function which given an argument in E build an element in any type A:

```
Coq < Variable A : Type.
A is assumed

Coq < Fixpoint Eany (x : E) : A :=
Coq <    match x with (Ei y) => Eany y end.
Eany is recursively defined (decreasing on 1st argument)
```

In particular one can prove `False` from an hypothesis $x : E$.

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
```

Exercises

- Define an `xor` function over booleans. Check the properties `xor true true = false`, `xor true false = true` and `xor 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:

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

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

4.3 Inductive Relations

Inductive definitions can be used to introduce relations specified by a set of closure properties (like inference rules or 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 are 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:

$$\forall n : \texttt{nat}, 0 \leq n \qquad \forall nm : \texttt{nat}, n \leq m \Rightarrow (\texttt{S}\,n) \leq (\texttt{S}\,m)$$

which is sometimes presented as a set of inference rules

$$\frac{}{0 \leq n} \qquad \frac{n \leq m}{(\texttt{S}\,n) \leq (\texttt{S}\,m)}$$

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).
LE is defined
LE_ind is defined
```

This declaration introduces identifiers `LE`, `LE_O` and `LE_S`, each having the type specified in the declaration. The `LE_ind` theorem is introduced which captures the minimality of the relation.

```
Coq < Check LE_ind.
LE_ind
     : forall P : nat -> nat -> Prop,
       (forall n : nat, P 0 n) ->
       (forall n m : nat, LE n m -> P n m -> P (S n) (S m)) ->
       forall n n0 : nat, LE n n0 -> P n n0
```

Actually, the definition of the order relation on natural numbers in Coq standard library is slightly different:

```
Coq < Print le.
Inductive le (n : nat) : nat -> Prop :=
    le_n : n <= n | le_S : forall m : nat, n <= m -> n <= S m
For le: Argument scopes are [nat_scope nat_scope]
For le_n: Argument scope is [nat_scope]
For le_S: Argument scopes are [nat_scope nat_scope _]
```

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. Both definitions of the order can be proved equivalent. In general there are multiple ways to define the same relation by inductive declarations (or possibly recursive functions). One has to keep in mind that they are different *implementations* of the same notion and that like in programming some of the choices will have consequences on the easiness of doing subsequent proofs with these notions.

Exercises.

- Define an inductive predicate `Natural` over $\mathbb{Z}$ which characterises `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.

4.4 Elimination of Inductive Definitions

Proof by case analysis: the `destruct` tactic. An object in an inductive definition I, when fully instantiated and evaluated will be formed after one of the constructors of I. When we have a term t in I, we can reason by case on the constructors the term t can be evaluated to using the `destruct` t tactic. This tactic generates a new subgoal by constructor and introduces new variables and hypothesis corresponding to the arguments of the constructor. COQ generates automatically names for these variables. It is recommended to use `destruct` t `as` *pat*; with *pat* a pattern for naming variables. *pat* will be written $[p_1|\ldots|p_n]$ with n the number of constructors of I. The pattern p_i will be written $(x_1,\ldots,x_k)$ if the constructor c_i expects k arguments.

If the goal has the form $\forall x : I, P$, then the tactic `intros` *pat*, will do the introduction of x and will immediately after destruct this variable using the pattern as in the following example:

```
Coq < Goal forall A B : Prop, (A /\ ~ B) \/ B -> ~A -> B.
1 subgoal

  ============================
   forall A B : Prop, A /\ ~ B \/ B -> ~ A -> B

Coq < intros A B [ (Ha,Hnb) | Hb ] Hna.
2 subgoals

  A : Prop
  B : Prop
  Ha : A
  Hnb : ~ B
  Hna : ~ A
  ============================
   B
subgoal 2 is:
 B

Coq < contradiction.
1 subgoal

  A : Prop
  B : Prop
  Hb : B
  Hna : ~ A
  ============================
   B

Coq < auto.
Proof completed.
```

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 bound in the goal. 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 generalises the dependent hypotheses of the expression on which induction applies.

Induction over Data Types. For an inductive type I, the induction scheme is given by the theorem *I_ind*; it generalises the standard 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 for the tactic `induction` *term* 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 tactic "`pattern` *term* `at` *occs*" can be applied first.

It is sometimes necessary to generalise the goal before performing induction. This can be done using the `cut` *`prop`* tactic, which changes the goal G into *prop* $\Rightarrow G$ and generates a new subgoal *prop*. If the generalisation involves some hypotheses, one may use the `generalise` tactic first (if `x` is a variable of type A, then `generalise` 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 $(I\ x_1 \ldots x_n) \Rightarrow G$ where I is the inductive relation. The goal G is abstracted w.r.t. $x_1 \ldots x_n$ to build the relation used in the induction. It works well when $x_1 \ldots x_n$ are either parameter of the inductive relation or variables. If some of the x_i are complex terms, the system may fail to find a well-typed abstraction or may infer a non-provable property.

If no recursion is necessary then the tactic `inversion` *`term`* is to be preferred (it exploits all informations in $x_1 \ldots x_n$). If recursion is needed then one can try to first change the goal into the equivalent one (assuming x_i is a non-variable, non-parameter argument):

$$\forall y, (I\ x_1 \ldots y \ldots x_n) \Rightarrow x_i = y \Rightarrow G$$

and then do the induction on the proof of $(I\ x_1 \ldots y \ldots x_n)$.

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

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

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

5 Solving some of the LASER 2011 Benchmark challenges

These are indications, partial or full solutions to some of the given challenges. It is the occasion to cover more advanced features of COQ from the user interface (notations) or from the theoretical side (dependent types) as well as to illustrate different styles of modeling problems in COQ.

We only present functional solutions to the problems, using mathematical objects.

5.1 Arithmetic

5.1.1 Absolute value

The challenge is about constructing and proving the absolute function on machine integers.

We show this challenge on mathematical integers. The COQ standard library contains a module for 31-bit integer arithmetic. The Compcert project[5] provides a library `http://compcert.inria.fr/src/lib/Integers.v` defining machine integers as mathematical numbers modulo 2^N. Both can be used as the basis for the full challenge.

Mathematical integers in COQ are defined as a type `Z`. They have a representation similar to the one of $\mathbb{Z}$ introduced before except that the positive part uses a binary representation (type `positive`) instead of the type `nat` of unary numbers.

```
Coq < Require Import ZArith.

Coq < Print Z.
Inductive Z : Set :=
    Z0 : Z | Zpos : positive -> Z | Zneg : positive -> Z
For Zpos: Argument scope is [positive_scope]
For Zneg: Argument scope is [positive_scope]
```

The library on integers introduces the same arithmetic notations available for natural numbers. One can force the interpretation in one category by using a suffix e%nat or e%Z and also define the default interpretation to be the one of integers using the command `Open Scope Z_scope`.

```
Coq < Check 0.
0
     : nat

Coq < Check 0%Z.
0%Z
     : Z

Coq < Open Scope Z_scope.

Coq < Check 0.
0
     : Z

Coq < Check 0%nat.
0%nat
     : nat
```

The absolute value function is part of COQ standard arithmetic library (function `Zabs`), and the expected result is a theorem named `Zabs_pos`). However we may define it more naively. We need a function which tests the sign of an integer. In COQ it is possible to mix programs and specifications. In particular, given two properties A and B, the type $\{A\} + \{B\}$ generalises both boolean values and logical disjunction. When given a proof b of $\{A\}+\{B\}$, one can build another program by case analysis like if b was a boolean value: **if** b **then** c_1 **else** c_2 as for booleans.

```
Coq < Require Import ZArith_dec.

Coq < SearchAbout ({_<=_}+{ _ }).
Z_le_dec: forall x y : Z, {x <= y} + {~ x <= y}
Z_le_gt_dec: forall x y : Z, {x <= y} + {x > y}
Zmin_le_prime_inf:
  forall n m p : Z, Zmin n m <= p -> {n <= p} + {m <= p}

Coq < Definition abs (z : Z) :  Z := if Z_le_dec 0 z then z else -z.
abs is defined
```

When reasoning on such a program (using `destruct`), we shall get an extra hypothesis (A or B) in each case.

```
Coq < Lemma abs_pos : forall z, 0 <= abs z.
1 subgoal

  ============================
   forall z : Z, 0 <= abs z

Coq < intro z; unfold  abs.
1 subgoal

  z : Z
  ============================
   0 <= (if Z_le_dec 0 z then z else - z)

Coq < destruct (Z_le_dec 0 z).
2 subgoals

  z : Z
  z0 : 0 <= z
  ============================
   0 <= z
subgoal 2 is:
 0 <= - z

Coq < trivial.
1 subgoal

  z : Z
  n : ~ 0 <= z
  ============================
   0 <= - z
```

The proof is completed using the `auto` tactic on the database `zarith` containing lemmas on arithmetic.

```
Coq < auto with zarith.
Proof completed.
```

5.1.2 Bank account

The challenge is to implement a class for an account with a balance represented as an IEEE floating point number and to specify a deposit method.

Of course the difficulty comes from the interpretation of the `plus` operation which will be a floating point number operation with rounding in the program ans possibly a more mathematical operation in the specification.

In COQ, it is possible to manipulate real numbers (library `Reals`, with arithmetic notations) and there are also external libraries dealing with IEEE floating point real numbers, the most recent one being `Flocq` [2].

```
Coq < Require Import Reals.

Coq < Open Local Scope R_scope.

Coq < Require Import Fappli_IEEE.
```

The type `binary32` represents a single precision (normalised) floating point number with its sign (a boolean), its mantissa (a positive binary number between 2^{23} and $2^{23} - 1$) and its exponent (between -126 and 126). COQ is able to compute with these numbers. We can also choose the rounding mode of the addition.

```
Coq < Print binary32.
binary32 = binary_float 24 128
     : Set

Coq < Check b32_plus.
b32_plus
```

```
      : mode ->
        binary_float 24 128 ->
        binary_float 24 128 -> binary_float 24 128
```

```
Coq < Print mode.
Inductive mode : Set :=
    mode_NE : mode
  | mode_ZR : mode
  | mode_DN : mode
  | mode_UP : mode
  | mode_NA : mode
```

The function B2R transforms a floating point into the corresponding real number. We introduce convenient notations.

```
Coq < Implicit Arguments B2R [prec emax].

Coq < Notation bin32 b m e :=
Coq <           (B754_finite 24 128 b m e (eq_refl true)).

Coq < Notation "2 ^ x" := (shift x 1) (at level 30) : positive_scope.

Coq < (* binary32 representation of 1, 2^(-23) and 2^(-24) *)
Coq < Definition b32_one : binary32 := bin32 false (2^23) (-23).

Coq < Definition b32_2_minus23 : binary32 := bin32 false (2^23) (-46).

Coq < Definition b32_2_minus24 := bin32 false (2^23) (-47).
```

We can now implement the deposit function and introduce the property corresponding to its correctness.

```
Coq < Definition deposit (olda amount:binary32) : binary32
Coq <       := b32_plus mode_NE olda amount.
deposit is defined

Coq < Definition deposit_correct olda amount : Prop :=
Coq <        B2R (deposit olda amount) = (B2R olda + B2R amount)%R.
deposit_correct is defined
```

We can now show correct and incorrect behaviours.

```
Coq < Lemma ex1: deposit_correct b32_one b32_2_minus23.
1 subgoal

  ============================
   deposit_correct b32_one b32_2_minus23

Coq < compute.
1 subgoal

  ============================
   8388609 * / 8388608 =
   8388608 * / 8388608 + 8388608 * / 70368744177664
```

The proof can be finished using the field tactic to reason on real numbers. The following case can be proved to be incorrect:

```
Coq < Lemma ex2: ~ (deposit_correct b32_one b32_2_minus24).
1 subgoal

  ============================
   ~ deposit_correct b32_one b32_2_minus24

Coq < compute; intro.
1 subgoal

  H : 8388608 * / 8388608 =
      8388608 * / 8388608 + 8388608 * / 140737488355328
  ============================
   False
```

The proof comes from the fact that we have an hypothesis $(x = x + y)$ with $y \neq 0$ but unfortunately it is not direct in COQ (the automation on real numbers is still rudimentary); we do not give the details here.

5.2 Algorithms on arrays

We represent arrays by lists.

```
Coq < Import Datatypes.

Coq < Require Import List.

Coq < Print list.
Inductive list (A : Type) : Type :=
    nil : list A | cons : A -> list A -> list A
For nil: Argument A is implicit and maximally inserted
For cons: Argument A is implicit
For list: Argument scope is [type_scope]
For nil: Argument scope is [type_scope]
For cons: Argument scopes are [type_scope _ _]

Coq < Open Scope Z_scope.

Coq < Open Scope list_scope.
```

Notations for lists include $a::l$ for the operator `cons` and l_1++l_2 for the concatenation of two lists.

5.2.1 Sum and maximum

Computing the sum and the maximum value of a list is done by a simple induction.

```
Coq < Fixpoint sum (l : list Z) : Z :=
Coq <    match l with nil => 0 | a::m => a + sum m end.
sum is recursively defined (decreasing on 1st argument)

Coq < Fixpoint max (l : list Z) : Z :=
Coq <    match l with nil   => 0
Coq <               | a::nil => a
Coq <               | a::m   => let b:= max m in if Z_le_dec a b then b else a
Coq <    end.
max is recursively defined (decreasing on 1st argument)
```

Because the pattern-matching for defining `max` is not elementary, it is useful to prove the corresponding equation to be used for rewriting.

```
Coq < Lemma max_cons : forall a m,
Coq <    m <> nil ->
Coq <    max (a::m) = let b:= max m in if Z_le_dec a b then b else a.

Coq < intro a; destruct m; trivial; intro H.

Coq < destruct H; trivial.

Coq < Qed.
```

We can after that enunciate the correctness property we want to prove:

```
Coq < Lemma sum_max_prop : forall l, sum l <= Z_of_nat (length l) * max l.
```

It will be proved by induction on l, then using the tactic `simpl` to do some of the simplifications on `sum` and `length` and then arithmetical reasoning.

correctness of max. To specify the behaviour of `max`, we could use the predicate `In` of the `List` library and say that whenever l is non empty then `max` l is in l and it is not less than all elements in l.

Correctness of `sum`. Our function `sum` satisfies the two equations:

$$\texttt{sum}\,\texttt{nil} = 0 \qquad \texttt{sum}\,(a::l) = a + \texttt{sum}\,l$$

which can be considered as a valid functional specification.

Termination. All functions in Coq terminate.

5.2.2 Linear search

With linear search of a zero in an array of non-negative integers, we go back to natural numbers.

```
Coq < Open Scope nat_scope.
```

In order to capture the special case where there is no 0 in the list, we prefer to use an option type with no or one value.

```
Coq < Print option.
Inductive option (A : Type) : Type :=
    Some : A -> option A | None : option A
For Some: Argument A is implicit
For None: Argument A is implicit and maximally inserted
For option: Argument scope is [type_scope]
For Some: Argument scopes are [type_scope _]
For None: Argument scope is [type_scope]
```

We use a recursive terminal definition:

```
Coq < Fixpoint linear (n:nat) (l:list nat) : option nat :=
Coq <       match l with nil => None
Coq <                  | a::m => if zerop a then Some n else linear (S n) m
Coq <       end.

Coq < Definition linear_search := linear 0.
```

In order to specify this function, it is convenient to introduce an inductive predicate `correct` such that `correct` k l is true when l starts with k non-zero elements and then contains a zero.

```
Coq < Inductive correct : nat -> list nat -> Prop :=
Coq <     correct_hd : forall a l, a=0 -> correct 0 (a::l)
Coq <   | correct_tl : forall a l n, a<>0 -> correct n l -> correct (S n) (a::l).

Coq < Hint Constructors correct.
```

The `Hint Constructors` command adds the constructors of the inductive definition in the hints database to be used by the `auto` tactic. Then the correctness of the function is the following lemma:

```
Coq < Lemma linear_correct : forall l n k,
Coq <       linear n l = Some k <-> (n <= k /\ correct (k-n) l).
```

which is proved by induction on l. The special case is a simple instantiation:

```
Coq < Lemma linear_search_correct :
Coq <    forall l k, linear_search l = Some k <-> correct k l.
```

The optimised case is a bit more tricky. First we can introduce an inductive definition for the limited decreasing property:

```
Coq < Inductive decrease : list nat -> Prop :=
Coq <    decrease_nil : decrease nil
Coq <  | decrease_cons : forall a b l,
Coq <       decrease (b::l) -> a <= S b -> decrease (a::b::l).

Coq < Hint Constructors decrease.
```

We shall use the function `skipn` from the `List` library which removes the first elements of a list. The definition we want looks like:

```
Coq < Fixpoint linear2 (n : nat) (l : list nat) : option nat :=
Coq <      match l with nil => None
Coq <                 | a::m => if zerop a then Some n
Coq <                           else linear2 (a+n) (skipn (a-1) m)
Coq <      end.
Error: Cannot guess decreasing argument of fix.
```

However, it is not accepted by COQ because there is no evident structural recursion. Actually this function terminates because the length of (`skipn` (a-1) m) is not greater than the one of m which is less than the one of l. So we have to move to a general recursion involving a well-founded ordering. COQ provides some definitions for that:

```
Coq < Check Fix.
Fix
     : forall (A : Type) (R : A -> A -> Prop),
       well_founded R ->
       forall P : A -> Type,
       (forall x : A, (forall y : A, R y x -> P y) -> P x) ->
       forall x : A, P x

Coq < Check Fix_eq.
Fix_eq
     : forall (A : Type) (R : A -> A -> Prop) (Rwf : well_founded R)
          (P : A -> Type)
          (F : forall x : A, (forall y : A, R y x -> P y) -> P x),
       (forall (x : A) (f g : forall y : A, R y x -> P y),
        (forall (y : A) (p : R y x), f y p = g y p) -> F x f = F x g) ->
       forall x : A,
       Fix Rwf P F x = F x (fun (y : A) (_ : R y x) => Fix Rwf P F y)
```

`Fix` is a general combinator for fixpoint definitions. Each time we do a recursive call, we have to provide a proof that the given element on the recursive call is less than the original one.

In COQ we have explicit proof terms that can be written explicitly in a program or we can use tactics for interactively building a computational term. None of these solutions is very convenient. COQ provides special tools to write programs containing logical parts but to solve these parts using tactic. This is the `Program` facility designed by M. Sozeau [7].

```
Coq < Require Export Program.

Coq < Program Fixpoint linear2 (n : nat) (l : list nat) {measure (length l)}
Coq <     : option nat
Coq <     := match l with nil => None
Coq <                   | a::m => if zerop a then Some n
Coq <                             else linear2 (a+n) (skipn (a-1) m)
Coq <        end.
```

We have one obligation to solve in order to make sure the recursive call decreases the measure. This property comes from the following lemma proved by induction on l:

```
Coq < Check skip_length.
skip_length
     : forall (A : Type) (n : nat) (l : list A),
       length (skipn n l) <= length l
```

We now solve the obligation:

```
Coq < Next Obligation.
1 subgoal
```

```
  n : nat
  a : nat
  m : list nat
  H : 0 < a
  linear2 : nat ->
            forall l : list nat,
            length l < length (a :: m) -> option nat
  ============================
   length (skipn (a - 1) m) < length (a :: m)

Coq < intros; apply le_lt_trans with (length m); simpl; auto with arith.
Proof completed.
```

If we want to prove the correctness of this program, one can proceed as before except that we will have to follow the definition scheme of the function, namely a well-founded induction, then a pattern-matching on l then a case analysis on the head value.

It is more convenient to do the proof while building the function, and the `Program` environment will help doing that.

The idea is to enrich the return type of the function with the property we expect using the Coq construction for $\{x : A|P\}$ which is the type of pairs (t,p) with t an object of type A and p a proof of $P[x \leftarrow t]$. This type is almost a subtype construction, except that in Coq an object t of type $\{x : A|P\}$ is not an object of type A, but there is a projection function from $\{x : A|P\}$ to A (the term $`t$ denotes the projection of t, and if we destruct t of type $\{x : A|P\}$ as (x,p) we get x of type A and p of type P.

We shall need the following properties of `decrease`:

```
Coq < Lemma decrease_skip : forall n l, decrease l -> decrease (skipn n l).

Coq < Lemma decrease_correct_skip :
Coq <   forall l, decrease l ->
Coq <   forall m n, n <= hd 0 l -> correct m (skipn n l) -> correct (n+m) l.

Coq < Lemma skip_correct :
Coq <   forall n l, correct n l ->
Coq <   forall m, m <= n -> correct m (skipn (n-m) l).
```

The fixpoint definition looks now like:

```
Coq < Program Fixpoint linear3 (n : nat) (l : list nat) {measure (length l)} :
Coq <    { res : option nat |
Coq <      decrease l -> forall k, res=Some k <-> (n<=k /\ correct (k-n) l) }
Coq <     := match l with nil => None
Coq <                   | a::m => if zerop a then Some n
Coq <                             else linear3 (a+n) (skipn (a-1) m)
Coq <        end.
```

It generates 4 proof obligations (correctness in the three branches and termination).

```
Coq < Obligations.
4 obligation(s) remaining:
Obligation 1 of linear3_func:
forall (n : nat) (l : list nat),
(forall (n0 : nat) (l0 : list nat),
 length l0 < length l ->
 {res : option nat |
 decrease l0 ->
 forall k : nat, res = Some k <-> n0 <= k /\ correct (k - n0) l0}) ->
[] = l ->
decrease [] ->
forall k : nat, None = Some k <-> n <= k /\ correct (k - n) [].

Obligation 2 of linear3_func:
forall (n : nat) (l : list nat),
(forall (n0 : nat) (l0 : list nat),
 length l0 < length l ->
 {res : option nat |
 decrease l0 ->
 forall k : nat, res = Some k <-> n0 <= k /\ correct (k - n0) l0}) ->
forall (a : nat) (m : list nat),
a :: m = l ->
a = 0 ->
decrease (a :: m) ->
forall k : nat, Some n = Some k <-> n <= k /\ correct (k - n) (a :: m).

Obligation 3 of linear3_func:
nat ->
forall l : list nat,
(forall (n0 : nat) (l0 : list nat),
 length l0 < length l ->
 {res : option nat |
 decrease l0 ->
 forall k : nat, res = Some k <-> n0 <= k /\ correct (k - n0) l0}) ->
forall (a : nat) (m : list nat),
a :: m = l -> 0 < a -> length (skipn (a - 1) m) < length l.

Obligation 4 of linear3_func:
forall (n : nat) (l : list nat)
  (linear3 : forall (n0 : nat) (l0 : list nat),
             length l0 < length l ->
             {res : option nat |
             decrease l0 ->
             forall k : nat,
             res = Some k <-> n0 <= k /\ correct (k - n0) l0})
  (a : nat) (m : list nat) (Heq_l : a :: m = l) (H : 0 < a),
decrease (a :: m) ->
forall k : nat,
`(linear3 (a + n) (skipn (a - 1) m)
    (linear3_func_obligation_3 n l linear3 a m Heq_l H)) =
Some k <-> n <= k /\ correct (k - n) (a :: m).
```

5.2.3 Sorting

There is a `Sorting` library in COQ with proofs of mergesort and heapsort.

5.2.4 Binary search

Binary search on lists is not so interesting, we suggest to implement and prove it using an array represented as a function from `nat` to a type A and the length of the array.

5.3 Linked lists

In order to represent structures with pointers, on need to exhibit a model for the memory. We can take $\mathbb{Z}$ for the set of addresses and add a special value for the null pointer. We enter in a mode with implicit arguments automatically computed.

```
Coq < Set Implicit Arguments.

Coq < Definition adr := option Z.
adr is defined

Coq < Definition null : adr := None.
null is defined
```

We define a node in a linked list as a record with a field for the value (here a natural number) and a next field with the address of the rest of the list.

```
Coq < Record node : Type := mknode { value : nat ; next : adr}.
node is defined
node_rect is defined
node_ind is defined
node_rec is defined
value is defined
next is defined
```

A record is a special case of inductive definition where there is only on constructor. The system derives automatically terms for the two projections `value` of type `node` $\rightarrow$ `nat` and `next` of type `node` $\rightarrow$ `adr`. Then the heap is a partial function from addresses to node which is represented as a total function from $\mathbb{Z}$ to `option node`. We

```
Coq < Definition heap := Z -> option node.
heap is defined

Coq < Definition val (h : heap) (a : adr) : option node
Coq <       := match a with None => None | Some z => h z end.
val is defined
```

We define the property for an object in an option type to be different of `None`.

```
Coq < Definition alloc A (a:option A) : Prop
Coq <       := match a with Some _ => True | None  => False end.
alloc is defined
```

It is equivalent to $a \neq$ `None` but defined in a computational way: a proof of `alloc` a will reduce either to `True` or `False`. We define another partial function for access but instead to output an optional type, it takes an extra argument as input which ensures the value exists.

```
Coq < Definition access (h:heap) (a:adr) : alloc (val h a) -> node
Coq <      := match (val h a) as x return alloc x -> node
Coq <                with None    => fun (H:False) => False_rect _ H
Coq <                   | Some n => fun (H:True)   => n
Coq <         end.
access is defined
```

We see an example of dependent pattern matching:

$$\textbf{match } t \textbf{as } x \textbf{ return } P \textbf{ with } p_1 \Rightarrow c_1 | \ldots p_n \Rightarrow c_n \textbf{ end}$$

The type of the **match** expression is $P[x \leftarrow t]$ and in each branch, x is substituted by the pattern.

In the first case we have to build an object in the type `alloc None` $\rightarrow$ `node` but because `alloc None` is equivalent to `False` this branch will never be accessed, so we provide a dummy element built form the proof of `False`.

In COQ all functions have to be total and terminating. If a list is cyclic or at some point an address is not allocated then the program will go wrong. So we introduce a predicate depending on an address and a heap which captures that following the links we always find allocated addresses until we reach the null address.

```
Coq < Inductive LList (h : heap) (a:adr) :  Prop :=
Coq <   mkLL : forall (LLa : alloc a -> alloc (val h a)),
Coq <           (forall (p:alloc a), LList h (next (access h a (LLa p))))
Coq <           -> LList h a.
```

It says that (`LList` h a) if whenever a is not null, it is allocated in the heap and the next address is itself a well-formed list. The strange form comes from the fact that the access function depends on a proof that the value in not `None`.

We easily derive the expected properties:

```
Coq < Lemma LL_null : forall h, LList h null.
```

```
Coq < Lemma LL_cons : forall h a (q:alloc (val h a)),
Coq <        LList h (next (access h a q)) -> LList h a.
```

We can also prove the other direction :

```
Coq < Lemma LL_alloc_val : forall h a, LList h a -> alloc a -> alloc (val h a).

Coq < destruct 1; trivial.

Coq < Defined.

Coq < Lemma LL_next : forall h a (L:LList h a) (p:alloc a),
Coq <           LList h (next (access h a (LL_alloc_val L p))).

Coq < unfold LL_alloc_val; destruct L; trivial.

Coq < Defined.
```

We use the keyword `Defined` instead of `Qed`. In COQ a constant can be defined as `Opaque` and will never be unfolded or reduced, which is the expected behaviour for most theorems. Or it can be declared as transparent. In this case, the proof of `LList` will be used inside `Coq` to control fixpoint definitions and need to be transparent, which is obtained with the `Defined` command.

Now, in order to build a function by following the links starting from an address a which corresponds to a well-formed list, we use a fixpoint that will be structurally decreasing on the proof of (`LList` h a).

We first introduce a program to test whether or not an address is `null`.

```
Coq < Definition nullp (a:adr) : {a=null}+{alloc a}.

Coq < destruct a; simpl; auto.

Coq < Defined.
```

As a first example, we build a logical list from a well-formed linked list.

```
Coq < Variable h : heap.
h is assumed

Coq < Fixpoint LL_list (a:adr) (La: LList h a) : list nat :=
Coq <      match nullp a with
Coq <        left  p => nil
Coq <      | right p => value (access h a (LL_alloc_val La p))
Coq <                   ::LL_list (LL_next La p)
Coq <      end.
LL_list is recursively defined (decreasing on 2nd argument)
```

If we want to prove the fixpoint equation, we need a case analysis in the proof of `LList`.

```
Coq < Lemma LL_list_eq : forall (a:adr) (La: LList h a),
Coq <       LL_list La = match nullp a with
Coq <        left  p => nil
Coq <      | right p => value (access h a (LL_alloc_val La p))
Coq <                   ::LL_list (LL_next La p)
Coq <      end.

Coq < destruct La; trivial.

Coq < Qed.
```

5.3.1 Linear search

Doing the naive linear search follows the same scheme:

```
Coq < Fixpoint LL_linear (a:adr) (La: LList h a) (n:nat) : option nat :=
Coq <    match nullp a with
Coq <        left  p => None
Coq <      | right p => if zerop (value (access h a (LL_alloc_val La p)))
Coq <                      then Some n
Coq <                      else LL_linear (LL_next La p) (S n)
Coq <    end.
```

It is possible to specify this program using the same predicate `correct` as before:

```
Coq < Lemma linear_correct : forall a (La:LList h a) n k,
Coq <       LL_linear La n = Some k <-> (n <= k /\ correct (k-n) (LL_list La)).
```

The proof will go by induction on the proof `La` of (`LList` h a) but because `LList` has type `Prop`, the induction principle automatically generated by Coq is not powerful enough.

```
Coq < Check LList_ind.
LList_ind
     : forall (h : heap) (P : adr -> Prop),
       (forall (a : adr) (LLa : alloc a -> alloc (val h a)),
        (forall p : alloc a, LList h (next (access h a (LLa p)))) ->
        (forall p : alloc a, P (next (access h a (LLa p)))) -> P a) ->
       forall a : adr, LList h a -> P a
```

We need a principle which allows to prove $\forall a\,(La : \texttt{LList}\,h\,a), P\;a\;La$.
There is a special command to derive this more powerful principle:

```
Coq < Scheme LList_indd := Induction for LList Sort Prop.
LList_indd is defined
LList_indd is recursively defined
```

Then the proof of the lemma starts with:

```
Coq < induction La using LList_indd; simpl; intros.
1 subgoal

  h : heap
  a : adr
  LLa : alloc a -> alloc (val h a)
  l : forall p : alloc a, LList h (next (access h a (LLa p)))
  H : forall (p : alloc a) (n k : nat),
      LL_linear (l p) n = Some k <->
      n <= k /\ correct (k - n) (LL_list (l p))
  n : nat
  k : nat
  ============================
   match nullp a with
   | in_left => None
   | right p =>
       if zerop (value (access h a (LLa p)))
       then Some n
       else LL_linear (l p) (S n)
   end = Some k <->
   n <= k /\
   correct (k - n)
     match nullp a with
     | in_left => []
     | right p => value (access h a (LLa p)) :: LL_list (l p)
     end
```

```
Coq < case (nullp a); intros.
2 subgoals

  h : heap
  a : adr
  LLa : alloc a -> alloc (val h a)
  l : forall p : alloc a, LList h (next (access h a (LLa p)))
  H : forall (p : alloc a) (n k : nat),
      LL_linear (l p) n = Some k <->
      n <= k /\ correct (k - n) (LL_list (l p))
  n : nat
  k : nat
  e : a = null
  ============================
   None = Some k <-> n <= k /\ correct (k - n) []
subgoal 2 is:
 (if zerop (value (access h a (LLa a0)))
  then Some n
  else LL_linear (l a0) (S n)) = Some k <->
 n <= k /\
 correct (k - n) (value (access h a (LLa a0)) :: LL_list (l a0))
```

The rest of the proof is quite similar to the proof using logical lists.

5.3.2 Concatenation

The program for concatenation will modify the heap. We first define updating functions. We use a function to test equalities on addresses.

```
Coq < Lemma adr_eq_dec : forall (a b:adr), {a=b}+{~a=b}.
```

```
Coq < Definition upd_node_next (n:node) (a:adr) := mknode (value n) a.

Coq < Definition upd_heap_next (h:heap) a b (p : alloc (val h a)) : heap :=
Coq <     fun z => if adr_eq_dec (Some z) a
Coq <              then Some (upd_node_next (access h a p) b)
Coq <              else h z.
```

Then the concatenation is defined by:

```
Coq < Fixpoint LL_concat (h:heap) (a:adr) (La:LList h a)
Coq <                       (p:alloc a) (b:adr) : heap :=
Coq <   match nullp (next (access h a (LL_alloc_val La p))) with
Coq <        left _  => upd_heap_next h a b (LL_alloc_val La p)
Coq <      | right q => LL_concat (LL_next La p) q b
Coq <   end.
```

However, proving the correctness of this program is much more involved. We first have to prove that if a and b are well-formed lists in the heap h, then they still are in the heap obtained after concatenation. But it is not true if they share some nodes. So it involves reasoning on separation properties of the heap.

It can be done just for the example or by building first a library which derives principles from separation logic.

This example uses dependent types, and objects depending on logical properties. With this style of programming, the proof scripts can quickly become unreadable. However tools like `Program` (that we introduced before) or Type classes can be used to hide this information and let the system infer the missing terms. However, the user has to be aware that the COQ kernel manipulates the fully expanded terms.

5.4 Needham-Schroeder Public Key protocol

The formalisation of Needham-Schroeder Public Key protocol in COQ was first experimented by D. Bolignano [3].

The modelling uses inductive definitions which model the exchanges. We have three agents A, B, I for Alice, Bob and the Intruder.

```
Coq < Inductive agent : Set := A | B | I .
```

A nonce is a secret that is generated by one agent to be shared with another, in or formalisation, they have two agents as parameter. The atomic messages are names of the agents, nonces, secret keys. a message can be encoded or combined with another.

```
Coq < Inductive message : Set :=
Coq <  Name : agent -> message
Coq <       | Nonce : agent*agent -> message
Coq <       | SK   : agent -> message
Coq <       | Enc  : message -> agent -> message
Coq <       | P    : message -> message -> message.
```

The assumptions are that every message sent is received by everybody. Alice and Bob follow the protocol but the intruder can transform the messages (pairing, unpairing, encoding with public keys, decoding when he knows the secret key).

We define three mutually inductive definitions:

- `send` which takes an agent and a message and implements the protocol rules plus the intruder capabilities;
- `receive` which takes an agent and a message and just says that everybody receive everything;
- `known` which characterises the knowledge of the intruder, some basic facts such as the name of the agents, his/her own secret key, plus the capability to eavesdrop the messages and massage them.

The protocol is parametrised by an agent X with which Alice starts the protocol.

```
Coq < Section Protocol.

Coq < Variable X:agent.

Coq < Inductive send : agent -> message -> Prop :=
Coq <      init   : send A (Enc (P (Nonce (A,X)) (Name A)) X)
Coq <    | trans1 : forall d Y,
Coq <               receive B (Enc (P (Nonce d) (Name Y)) B)
Coq <                -> send B (Enc (P (Nonce d) (Nonce (B,Y))) Y)
Coq <     | trans2 : forall d, receive A (Enc (P (Nonce (A,X)) (Nonce d)) A)
Coq <                -> send A (Enc (Nonce d) X)
Coq <     | cheat : forall m, known m -> send I m
Coq < with receive : agent -> message -> Prop :=
Coq <       link : forall m Y Z, send Y m -> receive Z m
Coq < with known : message -> Prop :=
Coq <       spy : forall m, receive I m -> known m
Coq <     | name : forall a, known (Name a)
Coq <     | secret_KI : known (SK I)
Coq <     | decomp_l : forall m m', known (P m m') -> known m
Coq <     | decomp_r : forall m m', known (P m m') -> known m'
Coq <     | compose  : forall m m', known m -> known m' -> known (P m m')
Coq <     | crypt  : forall m a, known m -> known (Enc m a)
Coq <     | decrypt : forall m a, known (Enc m a) -> known (SK a) -> known m.

Coq < End Protocol.
```

It is correct if the fact that B receives the acknowledgement (the nounce he generated for Alice) means that the protocol was initiated by Alice to talk with Bob. Also in that case, the nounces which are generated by Alice and Bob for each other should remain a shared secret. With this version, it is possible to prove that the protocol goes wrong, namely Alice starts the protocol with I and B gets the acknowledgement.

```
Coq < Lemma flaw : receive I B (Enc (Nonce (B,A)) B).

Coq < Lemma flawB : known I (Nonce (B,A)).
```

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. `http://www.labri.fr/perso/casteran/CoqArt/index.html`.

[2] Sylvie Boldo and Guillaume Melquiond. Flocq : Floats for coq. http://flocq.gforge.inria.fr/, 2011.

[3] Dominique Bolignano. An approach to the formal verification of cryptographic protocols. In *CCS '96 Proceedings of the 3rd ACM conference on Computer and communications security*, 1996.

[4] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. `http://adam.chlipala.net/cpdt/`.

[5] Xavier Leroy and al. Compcert: compilers you can formally trust. `http://compcert.inria.fr/`, 2010.

[6] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. University of Pennsylvania, 2011. `http://www.cis.upenn.edu/~bcpierce/sf/`.

[7] Matthieu Sozeau. Program-ing finger trees in Coq. In Ralf Hinze and Norman Ramsey, editors, *12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, pages 13–24, Freiburg, Germany, 2007. `http://www.lri.fr/~sozeau/research/publications/Program-ing_Finger_Trees_in_Coq.pdf`.