Introduction to the Coq Proof-Assistant for Practical Software Verification

Christine Paulin-Mohring

LRI, Univ Paris-Sud, CNRS, Orsay F-91405
INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893
christine.paulin@lri.fr

Abstract. This paper is a tutorial on using the Coq proof-assistant for reasoning on software correctness. It illustrates features of Coq like inductive definitions and proof automation on a few examples including arithmetic, algorithms on functional and imperative lists and cryptographic protocols.

Coq is not a tool dedicated to software verification but a general purpose environment for developing mathematical proofs. However, it is based on a powerful language including basic functional programming and high-level specifications. As such it offers modern ways to literally program proofs in a structured way with advanced data-types, proofs by computation, and general purpose libraries of definitions and lemmas.

Coq is well suited for software verification of programs involving advanced specifications like language semantics and real numbers. The Coq architecture is also based on a small trusted kernel, making possible to use third-party libraries while being sure that proofs are not compromised.

1 Introduction

1.1 What Is Coq ?

The proof assistant Coq is an environment for developing *mathematical* facts. This includes defining objects (integers, sets, trees, functions, programs ...); making statements (using basic predicates and logical connectives); and finally writing proofs.

The Coq *compiler* automatically checks correctness of definitions (well-formed sets, terminating functions ...) and correctness of proofs.

The Coq *environment* helps with: advanced notations; proof search; modular developments. It also provides program extraction towards languages like Ocaml and Haskell for efficient execution of algorithms and linking with other libraries.

Impressive examples have been done using Coq. They cover different areas. In pure mathematics, one can notice the Fundamental theorem of Algebra (every polynomial has a root in $\mathbb{C}$) developed at Nijmegen in the team of Barendregt [23] in 2000, and more recently the Feit-Thompson theorem on finite groups under the supervision of Gonthier at the joint INRIA-Microsoft research center [25]. Many

B. Meyer and M. Nordio (Eds.): LASER 2012, LNCS 7682, pp. 45–95, 2012.

interesting proofs combine advanced algorithms and non-trivial mathematics like the proof of the four-color theorem by Gonthier & Werner at INRIA and Microsoft-Research [24], a primality checker using Pocklington and Elliptic Curve Certificates developed by Théry et al. at INRIA [40] and more recently the proof of a Wave Equation Resolution Scheme by Boldo et al. [11]. Coq can also be used to certify the output of external theorem provers like in the work on termination tools by Contejean and others [17], or the certification of traces issued from SAT & SMT solvers done by Grégoire and others [2]. Coq is also a good framework for formalizing programming environments: the Gemalto and Trusted Logic companies obtained the highest level of certification (common criteria EAL 7) for their formalization of the security properties of the JavaCard platform [14]; Leroy and others developed in Coq a certified optimizing compiler for C (Leroy et al.) [29]. Barthe and others used Coq to develop Certicrypt, an environment of formal proofs for computational cryptography [7]. G. Morrisett and others also developed on top of Coq the Ynot library for proving imperative programs using separation logic [32]. These represent typical examples of what can be achieved using Coq. Coq might not be the best tool for proving your everyday routine code but is definitely useful when sophisticated data-structures, algorithms and specifications are involved and also as a general framework to design special platforms for software verification.

Related Systems. Coq is a proof assistant similar to HOL systems, a family of interactive theorem provers based on Church's higher-order logic including Isabelle/HOL [36], HOL4 [33], HOL-light [27]), PVS [35], ...

Unlike these systems, Coq is based on an *intuitionistic* type theory and is consequently closer to Epigram [31], Matita [4] and also Agda [18] and NuPrl [16]. All these systems have in common that functions are programs that can be computed and not just binary relations like in mathematics.

More Informations on Coq. The Coq web site is located at `coq.inria.fr`. It contains official distributions (multi-platforms), the reference manual and also libraries and user's contributions.

The so-called *Coq'art* book by Yves Bertot and Pierre Castéran [9] provides a full presentation of the Calculus of Inductive Constructions from the point of view of Interactive Theorem Proving and Program Development. The course by B. Pierce on software foundations [37] is available on-line, it uses Coq for modeling and reasoning on semantics of programming languages. The book by A. Chlipala [15] concentrates on programming with Coq and make intensive use of dependent types. We also recommend the course notes *Coq in a Hurry* by Y. Bertot [8] as an alternative quick introduction to the Coq system.

History. Information on the history of Coq can be found in the preface of the reference manual. The origin of the language, the *pure Calculus of Constructions* and its first implementation go back to 1984 and are due to Coquand & Huet [19]. The language and the environment were constantly extended afterward: universes and tactics (Coquand, 1985), program extraction (Paulin&

Werner, 1989 – Letouzey 2002), inductive definitions (Coquand & Paulin, 1989), co-inductive definitions (Giménez, 1995), advanced pattern-matching (Cornes, 1995 – Herbelin, 2002), coercions (Saibi, 1997), efficient computations (Barras, 2001 – Grégoire, 2002), Modules (Courant, 1998 – Chrząszcz, 2004 – Soubiran, 2010), tactic language (Delahaye, 2000), automated tactics (Crégut, Boutin, Pottier, Besson, Sozeau, ...), type classes (Sozeau, 2009)...

1.2 Coq Architecture

It is important to understand that Coq is based on a two levels architecture. There is a relatively small kernel based on a language with few primitive constructions (functions, (co)-inductive definitions, product types, sorts) and a limited number of rules for type-checking and computation. The same language is used to represent objects, functions, propositions and proofs.

On top of this kernel, Coq provides a rich environment to help designing theories and proofs offering mechanisms like user extensible notations, tactics for proof automation, libraries. This environment can be used and extended safely because ultimately any definition and proof is checked by a safe kernel.

As a Coq user, one might be interested by finding quickly the high-level constructions that will be helpful to solve a problem, but one cannot always escape understanding the underlying low-level language in order to be able to develop new functionalities and to better control how certain constructions work in certain circumstances.

The following example illustrates this two levels structure. When dealing with integers, the user will enter the notation `5=2+3` but internally the Coq kernel will be given the term:

```
@eq Z (Zpos (xI (xO xH))) (Zplus (Zpos (xO xH)) (Zpos (xI xH)))
```

which explicits the type `Z` of the components and also the binary encoding of the integers.

1.3 Program Verification in Coq

Coq can be used for program verification in different ways :

- One can express the property "*the program p is correct*" as a mathematical statement in Coq and prove it is correct. It requires to represent precisely the semantics of the program and can be hard, but the proof is guaranteed.
- One can develop a specific program analyzer (model-checking, abstract interpretation,...) in Coq, prove it correct and use it. It is a huge investment, but one gets automatically a result which is guaranteed for each program instance.
- One can represent the program p by a Coq term t and the specification by a type T such that $t : T$ (which is automatically checked) implies p is correct. It works well for functional (possibly monadic) programs.

– One can also use an external tool to generate proof obligations and then use Coq to solve obligations. It is a less safe approach (unless your generator of proof obligations is certified) but it can deal with specifications in undecidable fragments where full automation will not work.

1.4 Outline

In section 2, we introduce quickly the basic Coq commands and tactics for logical reasoning. In section 3, we develop the notion of inductive definitions (both types and relations) and illustrate them on a board example, simple search algorithms and proof of cryptographic protocols. In section 4, we show examples of advanced programming using Coq. In section 5, we discuss proof automation and in particular proofs by reflection. We conclude by a discussion on the merits of Coq with respect to other approaches.

2 Basics of Coq System

2.1 First Steps in Coq

Coq is an interactive system intended to build *libraries of definitions and facts*. As a logical system it implements higher-order logic including arithmetic. It also includes a basic *functional programming* language. The environment provides additional tools such as libraries that can be quickly loaded, tools to query the environment in order to find the appropriate definitions and lemmas, and various mechanisms to simplify notations.

We recommend following the examples in this paper while experimenting with the system. A widely used interface is `coqide` which is part of the Coq standard distribution. An alternative is to use `emacs` with the `proof-general` [5] library. Both environments offer similar functionalities. A main window contains Coq commands and is sequentially interpreted. When entering proof mode, another window displays the current status of the proof (unsolved goals). Finally, a window displays Coq output messages (including errors). The interface `coqide` has also a special *Queries* window which displays results of various printing commands.

All Coq commands end with a dot. Comments are written between `(* .. *)`. This document has been prepared using Coq `V8.3pl3`. The prefix `Coq <` is used in front of Coq inputs, and Coq output is printed afterwards on a new line. The prefix is omitted when the output is not displayed. The Coq code corresponding to the examples presented in this paper are available from the author web page.

2.2 Basic Terms

A Coq object in the environment has a *name* and a *type*. The **Check** *term* command takes a name (or more generally a term) as an argument; it checks it is well-formed and displays its type.

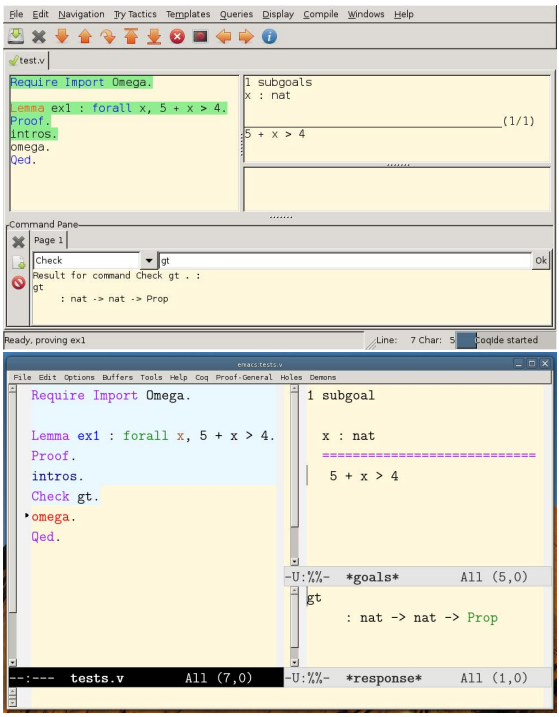


Fig. 1. Graphical interfaces for Coq: `coqide` (top), `Proof General` (bottom)

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

The object `nat` is a predefined type for natural numbers, its type is a special constant **Set** called a *sort*. The constant `0` has type `nat`. The object `S` is the successor function, it has type `nat` → `nat`. The binary function `plus` has type `nat` → `nat` → `nat` which should be read as `nat` → (`nat` → `nat`).

The term **fun** $x \Rightarrow t$ (or **fun** $(x : T) \Rightarrow t$ to indicate the type T of variable x) represents a function f such that $f(a) \equiv t[x \leftarrow a]$.

A function f can be applied to a term t using the notation $f\,t$. The term $f\,t_1\,t_2$ stands for $(f\,t_1)\,t_2$. The natural number `10` is just a notation for the successor function applied 10 times to `0` and the usual infix notation $t_1 + t_2$ can be used instead of `plus` $t_1\,t_2$.

```
Coq < Check (3+2).
3 + 2 : nat
```

The standard library defines the type of booleans `bool` with two inhabitants `true` and `false`. A choice on a boolean term b is written **if** b **then** t_1 **else** t_2.

Propositions. In Coq, logical propositions are also seen as terms. The type of propositions is the sort Prop. We present a summary of Coq syntax for logical propositions (first line presents paper notation and second line the corresponding Coq input).

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

The arrow represents implication, it associates to the right and `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 T (x y:T) (z:nat),P`
$\exists x, P$	`exists x, P`	`exists x:T, P`	*no multiple bindings*

The command **Check** verifies a proposition is well-formed but does not say if it is true or not.

```
Coq < Check (1+2=3).
1 + 2 = 3 : Prop
Coq < Check (forall x:nat, exists y, x=y+y).
forall x : nat, exists y : nat, x = y + y : Prop
```

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 command will usually also work with a more general type.

2.3 Logical Rules and Tactics

In order to establish that a proposition is true, we need to produce a proof. Following the approach introduced by R. Milner for the LCF system, we use backward reasoning with *tactics*. A *tactic* transforms a goal into a set of *subgoals* such that solving these subgoals is *sufficient* to solve the original goal. The proof succeeds when *no subgoals* are left.

In practice, we introduce a new goal in Coq using one of the following commands with *prop* representing a logical proposition (a well-formed term of type Prop).

Lemma *id* : *prop*. **Theorem** *id* : *prop*. **Goal** *prop*.

Coq implements a natural deduction logical system. Following Curry-Howard isomorphism, a proof of a proposition A is represented by a term of type A. So there is only one form of judgment $\Gamma \vdash p : A$. The environment Γ is a list of names associated with types $x : T$. When A is a type of objects, it is interpreted as "the term p is well-formed in the environment Γ and has type A". For instance $x : \texttt{nat} \vdash x + 1 : \texttt{nat}$. When A is a proposition, it is interpreted as "A is provable under the assumption of Γ and p is a witness of that proof". For instance $x : \texttt{nat}, h : x = 1 \vdash \ldots : x \neq 0$.

Axioms. The basic rule of natural deduction is the axiom rule when the goal to be proven is directly an hypothesis. The logical rule and corresponding tactics are:

$\dfrac{h : A \in \Gamma}{\Gamma \vdash h : A}$	`exact` h or `assumption`

Connectives. The rules for a connective are separated between *introduction rule(s)* giving a mean to prove a proposition formed with that connective if we can prove simpler propositions, and a rule of *elimination* which explains how we can use a proof of a proposition with that connective. In figure 2, we give the logical rule and the corresponding tactics. A tactic will work with a still unresolved goal, that we indicate using ? in place of the proof-term.

	introduction		elimination	
$\bot$			$\dfrac{\Gamma \vdash ? : \texttt{False}}{\Gamma \vdash ? : C}$	`exfalso`
$\neg$	$\dfrac{\Gamma, h : A \vdash ? : \texttt{False}}{\Gamma \vdash ? : \neg A}$	`intro` h	$\dfrac{\Gamma \vdash h : \neg A \quad \Gamma \vdash ? : A}{\Gamma \vdash ? : C}$	`destruct` h
$\rightarrow$	$\dfrac{\Gamma, h : A \vdash ? : B}{\Gamma \vdash ? : A \rightarrow B}$	`intro` h	$\dfrac{\Gamma \vdash h : A \rightarrow B \quad \Gamma \vdash ? : A}{\Gamma \vdash ? : B}$	`apply` h
$\forall$	$\dfrac{\Gamma, y : A \vdash ? : B[x \leftarrow y]}{\Gamma \vdash ? : \forall x : A, B}$	`intro` y	$\dfrac{\Gamma \vdash h : \forall x : A, B \quad \Gamma \vdash t : A}{\Gamma \vdash ? : B[x \leftarrow t]}$	`apply` h `with` $(x := t)$
$\wedge$	$\dfrac{\Gamma \vdash ? : A \quad \Gamma \vdash ? : B}{\Gamma \vdash ? : A \wedge B}$	`split`	$\dfrac{\Gamma \vdash h : A \wedge B \quad \Gamma, l : A, m : B \vdash ? : C}{\Gamma \vdash ? : C}$	`destruct` h `as` (l, m)
$\vee$	$\dfrac{\Gamma \vdash ? : A}{\Gamma \vdash ? : A \vee B}$ $\dfrac{\Gamma \vdash ? : B}{\Gamma \vdash ? : A \vee B}$	`left` `right`	$\dfrac{\Gamma \vdash h : A \vee B \quad \Gamma, l : A \vdash ? : C \quad \Gamma, l : B \vdash ? : C}{\Gamma \vdash ? : C}$	`destruct` h `as` $[l \mid l]$
$\exists$	$\dfrac{\Gamma \vdash t : A \quad \Gamma \vdash ? : B[x \leftarrow t]}{\Gamma \vdash ? : \exists x : A, B}$	`exists` t	$\dfrac{\Gamma \vdash h : \exists x : A, B \quad \Gamma, x : A, l : B \vdash ? : C}{\Gamma \vdash ? : C}$	`destruct` h `as` (x, l)
$=$	$\dfrac{t \equiv u}{\Gamma \vdash ? : t = u}$	`reflexivity`	$\dfrac{\Gamma \vdash h : t = u \quad \Gamma \vdash ? : C[x \leftarrow u]}{\Gamma \vdash ? : C[x \leftarrow t]}$	`rewrite` h

Fig. 2. Logical rules and corresponding tactics

It would be painful to apply only atomic rules as given in figure 2. Tactics usually combine in one step several introductions or elimination rules. The tactic `intros` does multiple introductions and infer names when none are given. The tactic `apply` takes as an argument a proof h of a proposition

$$\forall x_1 \ldots x_n, A_1 \rightarrow \cdots A_p \rightarrow B.$$

It tries to find terms t_i such that the current goal is equivalent to $B[x_i \leftarrow t_i]_{i=1\ldots n}$ and generates subgoals corresponding to $A_j[x_i \leftarrow t_i]_{i=1\ldots n}$. If some of the x_i are not infered by the system, it is always possible to use the variant `with` $(x_i := t_i)$.

Tactics associated with logical rules implement backward reasoning, but it is often useful to do forward reasoning, adding new facts in the goal to be proven. This is done using the `assert` tactic:

$\dfrac{\Gamma \vdash ? : B \quad \Gamma, h : B \vdash ? : A}{\Gamma \vdash ? : A}$	`assert` $(h : B)$

As an exercise, you may try to prove the following simple logical properties.

```
Lemma ex1: forall A B C:Prop,
                    (A -> B -> C) -> (A -> B) -> A -> C.
Lemma ex2: forall A, ~~~ A -> ~ A.
Lemma ex3: forall A B, A \/ B -> ~ (~ A /\ ~ B).
Lemma ex4: forall T (P:T -> Prop),
                    (~ exists x, P x) -> forall x, ~ P x.
```

Tactics can be combined using what is called a *tactical*:

t_1 `;` t_2	applies tactic t_1 then tactic t_2 on generated subgoals
t_1 `||` t_2	applies tactic t_1, when it fails, applies t_2
`try` t	applies tactic t, does nothing when t fails
`repeat` t	repeats tactic t until it fails

Some tactics are doing proof search to help solve a goal:

`contradiction`	when `False`, or both A and $\neg A$ appear in the hypotheses
`tauto`	solves propositional tautologies
`trivial`	tries very simple lemmas to solve the goal
`auto`	searches in a database of lemmas to solve the goal
`intuition`	removes the propositional structure of the goal then auto
`omega`	solves goals in linear arithmetic

Proving Equalities. The introduction rule of equality is reflexivity. In Coq, two terms t and u are *convertible* (written $t \equiv u$) when they represent the same value after computation. The elimination rule for equality allows to replace a term by an equal in any context. As a consequence, we have the following derived rules and corresponding tactics:

$\dfrac{\Gamma \vdash ? : u = t}{\Gamma \vdash ? : t = u}$	`symmetry`
$\dfrac{\Gamma \vdash ? : t = v \quad \Gamma \vdash ? : v = u}{\Gamma \vdash ? : t = u}$	`transitivity` v
$\dfrac{\Gamma \vdash ? : f = g \quad \Gamma \vdash ? : t_1 = u_1 \ldots \Gamma \vdash ? : t_n = u_n}{\Gamma \vdash ? : f\ t_1 \ldots t_n = g\ u_1 \ldots u_n}$	`f_equal`

Variants of the `rewrite` rule includes `rewrite <- H` when H is a proof of $t = u$ (or a generalization of it) which replaces u by t and the tactic `replace` u `with` t which does the replacement but also generates the goal $t = u$.

The `rewrite` tactics by default replaces 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`. When x is a variable and the context contains an hypothesis $x = t$ (or $t = x$) 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.

Finishing Proofs. The commands **Theorem** and **Lemma** are given a name *name* and a property A. They enter interactive proof mode in which tactics are used to transform the goal. Hopefully, after some effort there will be no remaining subgoals : the proof of A is finished. Actually, Coq is doing one more check before accepting the proof. From the tactics used, the system extracts a term p and the trusted kernel has to check that $\Gamma \vdash p : A$ is a valid judgment, which is done by elementary rules for type-checking p. This step is done with the commands **Qed** or **Save**. The proof is recorded in the environment and given the name *name* with type A. It can be used in other proofs like any hypothesis in the environment. It might seem useless to check again the proof, however, this choice of architecture allows to freely extend the set of tactics without compromising the safety of the proofs. Actually, some correctness checking (universes, well-formed definition of recursive functions) are not done when in interactive proof mode and consequently, it might be the case (in rare occasions) that a "finished proof" is actually not a correct proof.

If a proof is not finished, it is possible to admit an intermediate goal using the tactic **admit**. It will introduce an axiom corresponding to the current goal, it uses this axiom to solve the goal and continues with the next unproven goal. Using the command **Admitted** instead of **Qed** gives the possibility to end the proof, introducing the original goal A as an axiom. It is convenient to postpone a proof but it is also potentially dangerous. Assuming a false property might result in being able to prove $\bot$ and, consequently, everything becomes provable.

Safety in Coq is only guaranteed if there are no axioms left in the proof. The command **Print Assumptions** *name* can be used to display all axioms used in the theorem *name*.

Definitions. A new definition is introduced by the command:

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

The identifier *name* is then an abbreviation for the term *term*. The type *type* is optional as well as the arguments which are a list of identifiers possibly associated with types. For instance, the square function can be defined as follows:

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

A Coq definition *name* can be unfolded. The tactic `unfold` *name* replaces *name* with *term* it in the conclusion, `unfold` *name* `in` H does it in hypothesis H and finally all uses of *name* in the goal can be unfolded using tactic `unfold` *name* `in *`.

Variables and Axioms. 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* and **Hypothesis** *name* : *prop*

Several variables with the same type can be introduced with a single command, using the variant **Variables** 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 and no axioms are left.

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

```
Section test.
Variable A : Type.
Variables x y : A.
Definition double : A * A := (x,x).
Definition triple : A * A * A := (x,y,x).
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
Coq < Print triple.
triple = fun (A : Type) (x y : A) => (x, y, x)
     : forall A : Type, A -> A -> A * A * A
```

2.4 Libraries in Coq

The Coq environment is organized in a modular way. Some libraries are already loaded when starting the system. They can be displayed using the command:

```
Print Libraries.
```

Searching the Environment. The following interactive commands are useful to find information in libraries when doing proofs. They can be executed from the `coqide Queries` menu.

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

```
Coq < SearchAbout plus.
plus_n_O: forall n : nat, n = n + 0
plus_O_n: forall n : nat, 0 + n = n
plus_n_Sm: forall n m : nat, S (n + m) = n + S m
```

```
plus_Sn_m: forall n m : nat, S n + m = S (n + m)
mult_n_Sm: forall n m : nat, n * m + n = n * S m
nat_iter_plus :
  forall (n m : nat) (A : Type) (f : A -> A) (x : A),
  nat_iter (n + m) f x = nat_iter n f (nat_iter m f x)
```

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)
```

- `Check` *term*: checks if *term* can be typed and displays its type.
- `Print` *name*: prints the definition of *name* together with its type.
- `About` *name*: displays the type of the object *name* (plus other informations like qualified name or implicit arguments).

```
Coq < About pair.
pair : forall A B : Type, A -> B -> A * B
Arguments A, B are implicit and maximally inserted
<...>
Expands to: Constructor Coq.Init.Datatypes.pair
```

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 load-path, then it is loaded and opened (its contents is revealed).

The set of loaded modules and the load-path can be displayed with commands **Print Libraries** and **Print LoadPath.** The default load-path is the set of all sub-directories 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.

As usual in programming languages, the module names are used to organize the name space. The same "short name" can be used to represent different objects in different modules.

The command **Require** *name* (without **Import**) only loads the library, the objects inside are refered to by a qualified name: *dir.name.id*. The prefix *dir* is a logical name given to the directory of the loaded file. 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.

2.5 Examples

After all these generalities, we can try to do our first programs verification.

Absolute Value. One of the challenges proposed by the LASER summer school was to define an absolute value function on machine integers and prove the result is positive.

We first prove a much simpler result on mathematical integers. Mathematical integers in Coq are defined as a type `Z`. Their representation is based on a binary representation of positive numbers (type `positive`).

Definition and properties of integers are defined in the library `ZArith` that needs to be loaded first. In order to use the standard arithmetical notations for `Z`, we have to tell Coq to use them (otherwise Coq will interpret 0 or + as objects in `nat`).

```
Require Import ZArith.
Open Scope Z_scope.
```

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 want to do it naively. We need to test the sign of an integer so we need a boolean function for that. The command **SearchAbout** `(Z->bool)` shows there is a function `Zle_bool: Z -> Z -> bool`, then the command **SearchAbout** `Zle_bool` gives us several properties of this function, including:

```
Zle_cases: forall n m, if Zle_bool n m then n <= m else n > m
```

which links the boolean result of the `Zle_bool` function with the mathematical property. The proof goes as follows:

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

Coq < Lemma abs_pos : forall n, 0 <= abs n.
Coq < intro n; unfold abs.
1 subgoal
  n : Z
  ============================
   0 <= (if Zle_bool 0 n then n else - n)
Coq < assert (if Zle_bool 0 n then 0 <= n else 0 > n).
2 subgoals
  n : Z
  ============================
   if Zle_bool 0 n then 0 <= n else 0 > n
subgoal 2 is:
 0 <= (if Zle_bool 0 n then n else - n)
Coq < apply Zle_cases.
1 subgoal
  n : Z
  H : if Zle_bool 0 n then 0 <= n else 0 > n
  ============================
   0 <= (if Zle_bool 0 n then n else - n)
Coq < destruct (Zle_bool 0 n); auto with zarith.
Proof completed.
```

```
Coq < Qed.
abs_pos is defined
```

If we want to reason on machine integers, one can use the `Int31` library in Coq which represents 31-bit cyclic arithmetic, but because it is interpreted as positive integers between 0 and $2^{31}-1$, some work is needed before solving the problem and we shall not detail the proof here. The Compcert project [29] also provides a library `http://compcert.inria.fr/src/lib/Integers.v` defining machine integers as mathematical numbers modulo 2^N.

Bank Account. A second challenge was to implement a class for a bank 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 and possibly a 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` [12] (we use here version 2.0).

```
Require Import Reals Fappli_IEEE Fappli_IEEE_bits.
Open Local Scope R_scope.
```

The type `binary32` represents a single precision (normalized) floating point number with its sign (a boolean), its mantissa (a positive binary number between 0 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 : Type :=
    mode_NE | mode_ZR | mode_DN | mode_UP | mode_NA
```

The function `B2R` transforms a floating point into the corresponding real number. We introduce convenient notations for the numbers 1, 2^{-23} and 2^{-24}.

```
Notation bin32 b m e :=
(binary_normalize 24 128 gt0_24 gt_24_128 mode_NE m e b).

Definition b32_one := bin32 false (1) (0).
Definition b32_2_minus23 := bin32 false (1) (-23).
Definition b32_2_minus24 := bin32 false (1) (-24).
```

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

```
Definition deposit (olda amount:binary32) : binary32
     := b32_plus mode_NE olda amount.

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

We can now show correct and incorrect behaviors.

```
Coq < Lemma ex1: 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 proven to be incorrect:

```
Coq < Lemma ex2: ~ (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 fully automated in Coq (automation on real numbers is still rudimentary); we do not give the details here.

This example shows that Coq provides advanced libraries to reason on complex mathematical and algorithmic notions such as real and floating point numbers.

2.6 About Classical Logic

Coq implements an intuitionistic logic. Which means that $A \vee \neg A$ is not an axiom. Actually, both $A \vee B$ and $\exists x : A, B$ have a strong constructive meaning. Indeed, from a proof of $\vdash \exists x : A, B$ (without hypothesis), one can compute t such that $B[x \leftarrow t]$ is provable and from a proof of $A \vee B$ one can compute a boolean b and proofs of $b = \texttt{true} \rightarrow A$ and $b = \texttt{false} \rightarrow B$.

Classical reasoning is often not needed, manual proofs of $A \vee \neg A$ can be built for many simple formulas. It is also possible to use *classical versions* of logical connectives (negative formulas are classical). But if we really want to use classical logic, a library `Classical` introduces the excluded middle as an axiom.

3 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. We illustrate the use of inductive definitions on an example, modeling a game on a board containing bi-color tokens. We also study simple algorithms on lists and a cryptographic protocol.

3.1 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 declares 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 does not introduce inconsistency in the system.

The syntax for declaring an inductively defined type is:

$$\textbf{Inductive}\ name\ :\ sort\ :=\ c_1 : C_1 \mid \ldots \mid c_n : C_n.$$

where *name* is the name of the type to be defined; *sort* is one of Set or Type (or even Prop); c_i are the names of the constructors and C_i is the type of the 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 more complex terms expressing that *name* is the smallest set containing the terms build with constructors. These theorems provide induction principles to reason on objects in inductive types.

Examples. The data types 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
Coq < Check nat_ind.
nat_ind : forall P : nat -> Prop,
       P O -> (forall n : nat, P n -> P (S n))
       -> forall n : nat, P n
```

The type of booleans has only two constant constructors. The type of natural numbers has a constant `O` and a unary constructor `S` for the successor function. The product type is also inductively defined.

```
Coq < Print prod.
Inductive prod (A B : Type) : Type :=  pair : A -> B -> A * B
For pair: Arguments A, B are implicit and maximally inserted
```

We remark two things on this definition. First it is a polymorphic definition, parametrized by two types A and B. Second the constructor `pair` takes two arguments and pairs them in an object of type $A * B$, which is what is expected for a product representation.

Inductive Types and Equality. The constructors of an inductive type are injective and distinct. For instance one can prove `true` $\neq$ `false` and for natural numbers, $\mathtt{S}\,n = \mathtt{S}\,m \to n = m$ and $\mathtt{S}\,n \neq 0$. These lemmas are part of the standard library for natural numbers but have to be proven 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.

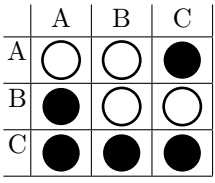
```
Coq < Goal (forall n, S (S n) = 1 -> 0=1).
Coq < intros n H.
1 subgoal
  n : nat
  H : S (S n) = 1
  ============================
   0 = 1
Coq < discriminate H.
Proof completed.
```

```
Coq < Goal (forall n m, S n = S (S m) -> 0 < n).
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. 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 degenerated case where there are no constructors. `True` is the proposition with only one proof `I` (corresponding to the `unit` type with only one constructor). Conjunction of two propositions corresponds to the product type and disjunction to an inductive proposition with two constructors. Existential quantifiers and equality are also inductively defined.

```
Coq < Print False.
Inductive False : Prop :=
Coq < Check False_ind.
False_ind : forall P : Prop, False -> P
Coq < Print or.
Inductive or (A B : Prop) : Prop :=
    or_introl : A -> A \/ B | or_intror : B -> A \/ B
Coq < Check or_ind.
or_ind : forall A B P : Prop,
         (A -> P) -> (B -> P) -> A \/ B -> P
```

The Board Example. The game we want to study involves nine bicolor tokens (one side black and one side white) which are placed on a 3×3 board.



At each step it is possible to choose one line or one column and to inverse the color of each token on that line or column. We want to study when a configuration is reachable from a starting configuration.

The data types involved in that example are the state of each token (black or white) which can be represented by a boolean or a special inductive type with two values. We need to identify a column and a line by a position (three possible values).

```
Inductive color : Type := White | Black.
Inductive pos : Type := A | B | C.
```

Finally we need to represent the board. It is convenient to represent it as three lines, each line being composed of three colors. In order to reuse functions, one can introduce a polymorphic type of triples of elements in an arbitrary type M (the definition is very similar to the definition of the product type).

```
Inductive triple M : Type := Triple : M -> M -> M -> triple M.
```

A line White/Black/White will be represented by the term `Triple White Black White`. The Coq kernel requires the type argument M to be explicitly given, so the Coq internal term is `Triple color White Black White`. However, the type `color` can be easily deduced from the type of `White` and can be systematically omitted in the input, thanks to the Coq declaration:

```
Set Implicit Arguments.
```

which tells Coq to infer type arguments whenever possible. Alternatively, the command **Implicit Arguments** *name* [*args*] can be used to force the implicit arguments of a given object *name*. On can also introduce a special notation for triples:

```
Notation "[ x | y | z ]" := (Triple x y z).
```

and define a function which given an element m in M builds a triple with the value m in the three positions.

```
Definition triple_x M (m:M) : triple M := [ m | m | m ].
```

3.2 Definitions by Pattern-Matching

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. This mechanisms is known in functional programming languages as *pattern-matching*. The Coq syntax is the following:

match *term* **with** $\mid c_1\ args_1 \Rightarrow term_1\ \ldots\ \mid c_n\ args_n \Rightarrow term_n$ **end**

In this construction, 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.

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

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

The board example. One can simply define a function which inverses a color:

```
Definition turn_color (c: color) : color :=
   match c with | White => Black | Black => White end.
```

On our board example, given a function f of type $M \to M$, one can define a function `triple_map` which given a triple (a, b, c), applies f to all components, and a function `triple_map_select` which also expects a position and applies the function f at that position.

```
Definition triple_map M f (t: triple M) : triple M:=
     match t with (Triple a b c) => [(f a)|(f b)|(f c)] end.
Definition triple_map_select M f p t : triple M :=
    match t with (Triple a b c) =>
      match p with | A => [ (f a) | b | c ]
                   | B => [ a | (f b) | c ]
                   | C => [ a | b | (f c) ]
      end
    end.
```

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

```
Definition nozero n m :=  match n,m with
          | O,_ => false |_,O => false |_,_ => true
          end.
```

However, the generalized 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 $\mathtt{c}_1$ and $\mathtt{c}_2$ (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**.

3.3 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, has an inductive type and if each recursive call is performed on a term which can be checked to be structurally smaller than x. The basic idea is that x will usually be the main argument of a `match` 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* $(x_1 : type_1)$ ... $(x_p : type_p)\{\texttt{struct}\ x_i\} : type$:= *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 $\{\text{struct } x_i\}$ is omitted, the system will try to infer an appropriate argument.

The type of *name* is `forall` $(x_1 : type_1) \ldots (x_p : type_p),\ type$. Occurrences of *name* in *term* must be applied to at least i arguments and the ith must be structurally smaller than x_i.

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 | O => m | S p => S (plus1 p m) end.
plus1 is recursively defined (decreasing on 1st argument)

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

Restrictions on Fixpoint Declarations. There are strong syntactic restrictions on the kind of definitions that are accepted, there should be one decreasing argument for each fixpoint, the following definition will not be accepted:

```
Coq < Fixpoint test (b:bool) (n m:nat) : bool
Coq <  := match (n,m) with
Coq <  | (O,_)     => true  | (_,O)     => false
Coq <  | (S p,S q) => if b then test b p m else test b n q
Coq <  end.
Error: Cannot guess decreasing argument of fix.
```

However, it is possible to define functions with more elaborated recursive schemes using higher order functions like the Ackermann function:

```
Coq < Fixpoint ack (n m:nat) {struct n} : nat
Coq <  := match n with
Coq <  |  O  => S m
Coq <  | S p => let fix ackn (m:nat) {struct m} :=
Coq <               match m with 0 => ack p 1
                                 | S q => ack p (ackn q)
                    end
Coq <           in ackn m
Coq < end.
ack is recursively defined (decreasing on 1st argument)
```

We may remark the internal definition of fixpoint using the **let fix** construction which defines the value of `ack` n as a new function `ackn` with one argument

and a structurally smaller recursive call. As an exercise, you may prove that the following equations hold using `reflexivity`.

```
Goal forall n, ack (S n) 0 = ack n 1.
Goal forall n m, ack (S n) (S m) = ack n (ack (S n) m).
```

Computation. A fixpoint can be computed only 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 just proven by reflexivity but requires a simple case analysis over the recursive argument.

```
Coq < Lemma plus1_eq : forall n m,
Coq <       plus1 n m
Coq <       = 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 such as:

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

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

```
Coq < Fixpoint Eany A x : A :=
      match x with (Ei y) => Eany A y end.
Eany is recursively defined (decreasing on 2nd argument)
```

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

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

```
Coq < Eval compute in (2 + 3)%nat.
     = 5%nat : nat
Coq < Eval compute in (turn_color White).
     = Black : color
Coq < Eval compute in
Coq <       (triple_map turn_color [Black|White|White]).
     = [White | Black | Black] : triple color
```

3.4 Algorithms on Lists

Another LASER challenge was proving algorithms on arrays. We choose to represent arrays by functional lists. We import the `List` and `ZArith` libraries and use the predefined notations. Notations for lists include $a::l$ for the operator `cons` and $l_1{+}{+}l_2$ for the concatenation of two lists.

```
Coq < Require Import List ZArith.
Coq < Open Scope Z_scope.
Coq < Open Scope list_scope.

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

Sum and Maximum. Computing the sum and the maximum value of a list is done by a simple fixpoint definition.

```
Fixpoint sum (l : list Z) : Z :=
   match l with nil => 0 | a::m => a + sum m end.

Fixpoint max (l : list Z) : Z :=
  match l with nil    => 0
             | a::nil => a
             | a::m   => let b:= max m in
                         if Zle_bool a b then b else a
  end.
```

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

```
Lemma max_cons : forall a m, m <> nil ->
  max (a::m) = let b:= max m in if Zle_bool a b then b else a.
intro a; destruct m; trivial; intro H.
destruct H; trivial.
Qed.
```

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

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

It is proven by induction on l (see section 3.6), then using the tactic `simpl` to do some of the simplifications on `sum` and `length` and then arithmetical reasoning.

Correctness of `max` and `sum`. To specify the behavior 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. Our function `sum` satisfies the two following equations which can be considered as a valid functional specification.

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

Termination. All functions in Coq terminate.

3.5 Inductive Relations

Inductive definitions can also 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. The type of this constructor is the logical formula associated to the clause. The syntax of such a definition is:

$$\textbf{Inductive}\ \mathit{name} : \mathit{arity} := c_1 : C_1 \mid \ldots \mid c_n : C_n.$$

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:

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

which can be alternatively be presented as a set of inference rules:

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

In Coq, such a relation is defined as follows:

```
Coq < Inductive LE : nat -> nat -> Prop :=
Coq < | LE_0 : 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_0` 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)
```

Given a natural number n, this definition characterizes the set of natural numbers which are not less than n as the smallest set which contains n and contains $\mathtt{S}\,m$ whenever it contains m. The parameter `(n:nat)` in the inductive definition of `le` is used to factor out `n` in the whole inductive definition. As a counterpart, the first argument of `le` must be `n` in the conclusion of each type of constructor. In particular, `n` could not have been a parameter in the definition of `LE` since `LE` must be applied to `(S n)` in the conclusion of the second clause. Both definitions of the order can be proven equivalent. For technical reasons, having more arguments as parameters in an inductive definition makes usually proofs easier. 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 these 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.

Examples

The Board Example. We have defined functions on triples and colors. We can now introduce the type of boards that will be a triple of lines, a line being a triple of colors. We also define two specific boards `start` and `target`. And we can define the functions `turn_col` and `turn_row` which inverse the colors.

```
Definition board := triple (triple color).

Definition start : board
    := [ [White | White | Black] |
         [Black | White | White] |
         [Black | Black | Black] ].

Definition target : board
    := [ [Black | Black | White] |
         [White | Black | Black] |
         [Black | Black | Black] ].

Definition turn_row (p: pos) : board -> board :=
   triple_map_select (triple_map turn_color) p.

Definition turn_col (p: pos) : board -> board :=
   triple_map (triple_map_select turn_color p).
```

Now if we want to define the relation between two boards corresponding to one step (inversing one line or one column), we can use predefined logical connectives:

```
Definition move1 (b1 b2: board) : Prop :=
    (exists p : pos, b2=turn_row p b1)
\/  (exists p : pos, b2=turn_col p b1).
```

or alternatively a direct inductive definition:

```
Inductive move (b1:board) : board -> Prop :=
  move_row : forall (p:pos), move b1 (turn_row p b1)
| move_col : forall (p:pos), move b1 (turn_col p b1).
```

If we want to define reachability, we need to consider the reflexive-transitive closure of the `move` relation. This is done easily with an inductive definition:

```
Inductive moves (b1:board): board -> Prop :=
  moves_init : moves b1 b1
| moves_step : forall b2 b3,
   moves b1 b2 -> move b2 b3 -> moves b1 b3.
```

One can prove simple properties like:

```
Coq < Lemma move_moves : forall b1 b2, move b1 b2 -> moves b1 b2.
Coq < intros; apply moves_step with b1; trivial.
1 subgoal
  b1 : board
  b2 : board
  H : move b1 b2
  ============================
   moves b1 b1
Coq < apply moves_init.
Proof completed.
```

We can prove that the board `target` is accessible from the board `start`.

```
Lemma reachable : moves start target.
apply moves_step with (turn_row A start); auto.
replace target with (turn_row B (turn_row A start)); auto.
Qed.
```

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

```
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 none or one value.

```
Coq < Print option.
Inductive option (A : Type) : Type :=
    Some : A -> option A | None : option A
```

We use a terminal recursive definition:

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

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.

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

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.

3.6 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 an arbitrary term t in I, we can reason by case on the constructors the term t can be evaluated to. The `destruct` t tactic generates a new subgoal for each 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. A pattern *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.
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.
```

For instance, the equivalence between the two definitions of one-step *move* in the board example can be easily done using the `destruct` tactic.

```
Lemma exboard : forall b1 b2, move1 b1 b2 <-> move b1 b2.
```

The `destruct` tactic, when applied to an hypothesis will clear this hypothesis from the goal. The `case` tactic is a more elementary tactic corresponding to the logical elimination rule when more control is needed.

Proof by Induction. The tactic to perform proofs by induction is written `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 an elimination rule for a logical connective or a minimality principle for an inductive relation. More precisely, an induction is the application of one of the principles which are automatically generated when the inductive definition 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 a natural number and the hypothesis we want to eliminate is the *num*-th unnamed hypothesis in the conclusion.

The `induction` tactic generalizes the dependent hypotheses of the expression on which the induction applies.

Induction over Data Types. For an inductive type I, the induction scheme is given by the theorem I_ind; it generalizes the standard induction over natural numbers. The main difficulty is to tell the system what is the property to be proven by induction. The default (inferred) property for 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 generalize the goal before performing induction. This can be done using the `cut` *prop* tactic, which changes the goal G into $prop \to G$ and generates a new subgoal *prop*. If the generalization 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` $x : A,\ G$).

The correctness of the `linear` function introduced earlier can be expressed by the following lemma:

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

which is proven by induction on l. The special case is a simple consequence when $n = 0$:

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

Induction over Proofs. If *term* belongs to an inductive relation then the induction tactic corresponds to the use of the minimality principle for this relation. Generally speaking, the property to be proven 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 well when $x_1 \dots x_n$ are either parameters 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 $x_1 \dots x_n$). 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):

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

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

For the board example, the transitivity of the `moves` relation (an arbitrary number of simple moves) is easily done by an induction on the proof of (`moves` $b_2\ b_3$).

```
Lemma moves_trans : forall (b1 b2 b3:board),
    moves b1 b2 -> moves b2 b3 -> moves b1 b3.

induction 2.
```

A more complex result will be to prove that there are no possible moves from the board `start` to the board with only white tokens (called `wboard`). So we have to prove:

```
Coq < Lemma not_reachable : ~ moves start wboard.
Coq < intro.
1 subgoal
  H : moves start wboard
  ============================
   False
```

After an introduction, we end up proving $\bot$ from the assumption `(moves start wboard)`. If we try an induction on this proof, the first goal will be to prove $\bot$ without any assumption, this is hopeless. A solution is to find an appropriate invariant of the game and to derive a contradiction from the assumption that `wboard` satisfies this invariant. The hint is to look at the number of white tokens at the 4 corners.

3.7 Advanced Inductive Definitions

The inductive definition mechanism of Coq is quite general and allows to model more than just algebraic data-types.

For instance, it is possible to represent trees with infinite branching, like in this type of ordinal notations.

```
Inductive ord := zero : ord | succ : ord -> ord
             | lim : (nat -> ord) -> ord.
```

The constructor `lim` takes a function as argument, for each natural number, it gives a new object of type `ord` which is a subterm.

Another important inductive definition is the accessibility property. Given a binary relation R on a type A, an element $x : A$ is accessible if there is no infinite chain $(x_i)_{i\in\mathbb{N}}$ such that $x_0 = x$ and $\forall i, R(x_{i+1}, x_i)$. Inductively, it is possible to define x to be accessible when all y such that $R(y, x)$ are also accessible. This is captured in the following definition:

```
Coq < Print Acc.
Inductive Acc A  (R:A -> A -> Prop) (x:A) : Prop :=
    Acc_intro : (forall y:A, R y x -> Acc R y) -> Acc R x
```

This inductive definition is the key for the definition of general fixpoints (see section 4.2)

Dependent types. It is also possible to introduce types indexed by other objects (also called dependent types) like in the type of vectors of size n:

```
Inductive vect (A:Type) : nat -> Type :=
  v0 : vect 0
| v1 : forall n, A -> vect n -> vect (S n).
```

This definition looks like the definition of lists but with an extra argument which will correspond to the size of the vector. Dependent types can also be defined recursively:

```
Fixpoint prodn A (n:nat) : Type :=
  match n with 0 => A | S n => A * prodn n end.
```

In that definition, the type (`prodn` A 2) is convertible with the type $A * (A * A)$.

3.8 Needham-Schroeder Public Key Protocol

The Needham-Schroeder Public Key protocol is intended to provide mutual authentication between two parties communicating on an insecure network using trusted public keys. The first published version was insecure. Its abstract version is given by the following exchange:

$A \longrightarrow B : \{N_A, A\}_{K_B}$	A sends to B a random number N_A and its name, encrypted using B public key
$B \longrightarrow A : \{N_A, N_B\}_{K_A}$	B decrypts A message and sends it back, adding a new random number N_B using A public key
$A \longrightarrow B : \{N_B\}_{K_B}$	A acknowledges decryption of B message

The formalization of this protocol in Coq was first experimented by D. Bolignano [13]. Inductive definitions are used to model the exchanges. We have three agents A, B, I for Alice, Bob and the Intruder.

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

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

```
Inductive message : Set :=
        Name : agent -> message
      | Nonce : agent * agent -> message
      | SK   : agent -> message
      | Enc : message -> agent -> message
      | 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 capability to send whatever message he/she knows;
- `receive` which takes an agent and a message and just says that everybody receive everything;
- `known` which characterizes 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 parametrized by an agent X with which Alice starts the protocol.

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

```
| decrypt : forall m a,
            known (Enc m a) -> known (SK a) -> known m.

End Protocol.
```

The protocol is correct if the fact that Bob receives the acknowledgment (the nonce he generated for Alice) means that the protocol was initiated by Alice to talk with Bob. Also in that case, the nonces 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 the intruder and Bob gets the acknowledgment.

```
Lemma flaw : receive I B (Enc (Nonce (B,A)) B).
Lemma flawB : known I (Nonce (B,A)).
```

In order to avoid the attack, it is sufficient for Bob to add its name to the nonces when he answers to Alice.

4 Functional Programming with Coq

In this section, we show how to represent in Coq programming constructions more elaborated than the "mathematical" functions we defined earlier.

4.1 Partiality and Dependent Types

An object of type $A \to B$ in Coq is a total function. Given a value of type A, the evaluation always terminates and gives a value of type B. If the function we want to implement is partial, there are several possibilities:

- choose an arbitrary value in B to extend the function. This solution does not work for polymorphic functions when B is a type variable because Coq types might be empty.
- use an option type to represent 0 or 1 value. The main drawback is that the function has now a type $A \to \texttt{option}\, B$ so case analysis is needed when using this function. This can be partially hidden using monadic notations.
- Consider the function as a relation F of type $A \to B \to \mathsf{Prop}$. We have then to prove the functionality of the relation (at most one output), each time we want to mention $f(x)$, we shall introduce a variable y and an hypothesis $F\, x\, y$. The relation also does not carry a priori an algorithm to compute the value of the function.
- Introduce an extra logical argument like explained in next paragraph.

Introducing Logic in Types. The Coq language allows to mix freely types and properties.

For instance, it is possible to add an explicit precondition to a function. Assuming our function f is only defined on a domain `dom`, we can define it as:

$$f : \forall x : A, \texttt{dom}\, x \to B$$

Each call to $f\,a$ requires a *proof* p of $\mathtt{dom}\,a$ and will be internally represented as: $f\,a\,p$. We can *partially hide* the proof in a subset type: $f : \{x : A | \mathtt{dom}\,x\} \to B$. Internally, the call $f\,a$ is represented by the term $f\,(a, p)$. The Coq definition of the subset type is:

```
Print sig.
Inductive sig (A : Type) (P : A -> Prop) : Type :=
    exist : forall x : A, P x -> sig P
```

High-level tools like ***Program*** and *type classes* help separate programming from solving proof obligations. Such that the user notations will remain close to ordinary programming.

Using Subset Types for Specifications. A proposition can also be used to restrict the image of a function, like in:

$$S : \mathtt{nat} \to \{n : \mathtt{nat} | 0 < n\}$$

The restriction can depend on the input:

$$next : \forall n : \mathtt{nat}, \{m : \mathtt{nat} | n < m\}$$

Other useful types with logical components are the constructive unions. Objects in `sumbool` are like booleans but a boolean value coming with the evidence that a property (A for `true` and B for `false`) is true.

```
Coq < Print sumbool.
Inductive sumbool (A B : Prop) : Set :=
    left : A -> {A} + {B} | right : B -> {A} + {B}
```

For instance the following property expresses the fact that the order on natural numbers is decidable:

```
Check forall n m:nat, {n <= m}+{m < n}.
```

Objects in `sumor` are like option values with the `None` case being associated with a property. It is useful in programs which have an exceptional case where no value is computed but some information on the input is established. For instance the following property expresses the fact that for any natural number n, one can find a number m smaller than n or one can establish that n is 0.

```
Check forall n, { m | m < n }+{ n=0 }.
```

Annotated Programs. The advantage of annotated programs is that both program and proof are developed simultaneously. The specification is available each time the program is used. The drawback is that inside Coq, the program contains proof-terms: printing, checking equality, reduction can become awful. Coq currently misses a systematic proof-irrelevance mechanism that will remove internally the dependency of terms with respect to proofs. A system like PVS for instance completely ignore proof-terms. In HOL based systems, it is not possible to build types dependent on proofs. In Coq, the extraction mechanism [30] allows to remove the part in programs related to logical proofs and convert the resulting term in an equivalent Ocaml or Haskell program.

4.2 Well-Founded Induction and Recursion

We have seen that primitive fixpoint definitions in Coq are only based on a simple structural recursion. So we need more sophisticated ways to define general algorithms. Typical examples are loops:

$$\textbf{let}\, f\, n = \textbf{if}\, p\, n\, \textbf{then}\, n\, \textbf{else}\, f\,(n+1)$$

or general well-founded fixpoints

$$\textbf{let}\, f\, x = t(x, f)$$

where any call to $f\, y$ in t is such that $y < x$ for a *well-founded* relation (no infinite decreasing sequence).

The main trick is to introduce an inductive definition which captures the termination argument and to do the structural recursion on that argument.

Loops. For instance, the loop construction only terminates when there exists $m \geq n$ such that $p\, m = \texttt{true}$. One can define inductively a predicate `Event` on natural numbers that captures the fact that the property P is true now or in the future.

```
Inductive Event P (n : nat) : Prop :=
   Now : P n -> Event P n
 | Fut : Event P (S n) -> Event P n.
```

Now we can prove that if P is not true now at time n, then it will be true somewhere in the future of time $n+1$.

```
Lemma Event_next P n : ~ P n -> Event P n -> Event P (S n).
```

This is done by case analysis on the proof p of `Event` $P\, n$. The first case $p = \texttt{Now}\, q$ is eliminated by contradiction because we have both $P\, n$ and $\neg P\, n$. The second case $p = \texttt{Fut}\, q$ is trivial because q is a proof of `Event` $P\, (\texttt{S}\, n)$. We remark that the proof we obtain for `Event` $P\, (\texttt{S}\, n)$ is a subterm of the original proof of `Event` $P\, n$.

```
Coq < Print Event_next.
Event_next =
fun (P : nat -> Prop) n (notP : ~ P n) (e : Event P n) =>
match e with
| Now H => match notP H return (Event P (S n)) with end
| Fut H => H
end
: forall P n, ~ P n -> Event P n -> Event P (S n)
```

To construct the loop function, we need the P predicate to be decidable. The following fixpoint construction is valid in Coq.

```
Fixpoint findP P (dec:forall n,{P n}+{~P n}) n (e:Event P n)
 : nat
 := match dec n with
     left _       => n
   | right notPn => findP dec (n:=S n) (Event_next notPn e)
   end.
```

The extraction mechanism gives us back exactly the function we wanted to write.

```
Coq < Extraction findP.
(* val findP : (nat -> sumbool) -> nat -> nat *)
let rec findP pdec n =
  match pdec n with
  | Left -> n
  | Right -> findP pdec (S n)
```

Well-Founded Recursion. A more general way to define a function using fixpoints is to rely on a well-founded ordering. We want to introduce a function $f : A \to B$ satisfying the equation:

$$\textbf{let}\ f\ x = t(x, f)$$

and to ensure that recursive calls are done on smaller instances, we may require the term t to have type $t : \forall x : A, (\forall y : A, y < x \to B) \to B$. We may actually generalize the type of f to cover dependent types and have $f : \forall x : A, P(x)$. The term t will have type: $\forall x : A, (\forall y : A, y < x \to P(y)) \to P(x)$. In Coq libraries, a combinator for well-founded fixpoint is predefined, and the fixpoint equation is proven.

```
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
```

Under the appropriate hypotheses, the fixpoint equation `Fix_eq` states:

```
  forall x, Fix Rwf P F x = F x (fun (y : A) (_ : R y x) => Fix Rwf P F y)
```

The fixpoint is on the proof of well-foundness of the relation, which is itself a proof that every object in A is accessible for the relation R.

The Optimized Linear Search. In this problem the array has an extra property that successive elements do not decrease by more than one (but they may increase arbitrarily). Now the linear search can be changed to a sub-linear search: if the first element a of the list in non-zero then instead to consider the rest of the list, on can skip a elements.

First we can introduce an inductive definition for the limited decreasing property:

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

We shall use the function `skipn` from the `List` library which, given a natural number n and a list l, removes the first n elements of l. The definition we want looks like:

```
Coq < Fixpoint linear2 n (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 $(\texttt{skipn}\,(a-1)\,m)$ 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 (a simple measure given by the length of the list in that case).

Coq provides special tools to write programs containing logical parts while solving these parts using tactics. This is the `Program` facility designed by M. Sozeau [38].

```
Require Export Program.

Program Fixpoint linear2 n (l:list nat) {measure (length l)}
 : option nat
 := match l with nil => None
              | a::m => if zerop a then Some n
                        else linear2 (a+n) (skipn (a-1) m)
    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 proven by induction on l:

```
Coq < Check skip_length.
skip_length : forall A n (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;
Coq < 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 also help doing that. We enrich the return type of the function with the property we expect using the Coq construction for $\{x : A|P\}$.

We shall need the following properties of `decrease`:

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

The fixpoint definition looks now like:

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

It generates 4 proof obligations (correctness in the three branches and termination) that can be displayed using the command **Obligations**.

Other Examples. Direct functional programming in Coq has been used for the development of quite impressive programs including compilers, static analyzers, kernels of the Coq system itself or the SMT solver alt-ergo. All these examples are related to symbolic computation which is not surprising. The existence of primitive inductive definitions in Coq facilitates the definition of abstract syntax trees or the definition of language semantics using inference rules. These programs will be much more complicated to develop in more traditional programming languages and more difficult to prove using first-order logic.

4.3 Imperative Programming

Coq embeds a pure functional language. However, it is possible to capture non functional behaviors using monadic constructions like in Haskell for instance. First, we introduce for each type A, a type `comp` A which represents computations leading eventually a value of type A. Then we need two standard functions `return` and `bind`. The function `return` (aka `unit`) has type $A \to$ `comp` A and `return` v represents the value v seen as the result of a computation; the function `bind` has type `comp` $A \to (A \to$ `comp` $B) \to$ `comp` B, `bind` passes the result of the first computation to the second one. We shall use the syntax: "`p <- e1; e2`" for `bind` e_1 (**fun** $p \Rightarrow e_2$). The monadic construction can be integrated in Coq using the type classes mechanism.

```
Coq < Class Monad (comp : Type -> Type) : Type :=
Coq <  {ret  : forall {A}, A -> comp A;
Coq <   bind : forall {A B},
Coq <          comp A -> (A -> comp B) -> comp B
Coq <  }.
ret is defined
bind is defined
```

Given a type transformer `comp`, the type class `Monad comp` encapsulates the two operators `return` and `bind`. In general, there will be one instance of `Monad` for each different `comp` operator. The type class mechanism is useful to share the same notations between different structures. Given the `comp` operator, the system will be looking automatically for an object in `Monad comp`.

```
Notation "X <- A ; B" := (bind A (fun X => B))
   (at level 30, right associativity).

Notation "'return' t" := (ret t) (at level 1).
```

We can deduce the `join` operator of monads in a generic way:

```
Definition join comp {_:Monad comp} A (t:comp (comp A))
     : comp A := X <- t; X.
```

The monad of errors can be defined using an option like type definition.

```
Inductive Error A := Raise:Error A | Val:A->Error A.
Implicit Arguments Raise [[A]].

Definition bindError A B (a:Error A) (f:A->Error B) : Error B
     := match a with | Raise => Raise | Val a => f a  end.

Instance ErrMonad : Monad Error
    := {ret := Val; bind := @bindError}.
```

One can use this monadic construction to define naturally a map function possibly raising errors:

```
Fixpoint mape A B (f:A->Error B) l : Error (list B) :=
  match l with | nil => return nil
               | (a::m) => b <- f a; mb <- mape f m;
                           return (b::mb)
  end.
```

Similarly, a state monad can be defined to simulate references.

```
Definition State st A := st -> A * st.
```

Similar approaches works for non-deterministic programs, probabilistic programs (see [6]), and other constructions.

4.4 The Linked Lists Example

Memory Representation. The previous monadic approach for states does not consider aliases in programs. If we need to deal explicitly with possible sharing we can introduce a functional representation of memory and addresses.

This can be done to model linked lists. We can take $\mathbb{Z}$ for the set of addresses and add a special value for the null pointer. 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. A record is a special case of inductive definition where there is only one 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`.

```
Coq < Definition adr := option Z.
Coq < Definition null : adr := None.

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

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

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

The state of the program will be the heap.

An alternative representation of memory would be to introduce more static separation (model à la Burstall-Bornat) by integrating the fact that the fields `value` and `next` will never be aliased, so we can consider that we have a different memory for each field, we keep separately a table `h` of allocated addresses. The state of the program consists of these three objects.

```
Definition value_m := Z -> nat.

Definition next_m := Z -> adr.

Definition val_m (h:Z->bool)(vm:value_m)(nm:next_m)(a:adr)
 : option node  := match a with
    | None   => None
    | Some z => if h z then Some (mknode (vm z) (nm z))
                else None
                   end.
```

With any of these models, we can specify logically the validity of addresses, and absence of aliases. If needed, high-level rules such as the one of separation logic can be encoded like in the Ynot Coq library to reason on imperative programs [32].

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

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

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.

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

We see here 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 corresponding 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 from 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.

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

It says that (`LList` h a) holds 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 is not `None`. We easily derive the expected properties:

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

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

We can also prove the other direction :

```
Lemma LL_alloc_val : forall h a,
      LList h a -> alloc a -> alloc (val h a).
destruct 1; trivial.
```

```
Defined.

Lemma LL_next : forall h a (L:LList h a) (p:alloc a),
          LList h (next (access h a (LL_alloc_val L p))).
unfold LL_alloc_val; destruct L; trivial.
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 behavior 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 needs to be transparent, which is obtained with the `Defined` command at the end of the proof.

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`.

```
Definition nullp (a:adr) : {a=null}+{alloc a}.
destruct a; simpl; auto.
Defined.
```

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

```
Variable h : heap.

Fixpoint LL_list (a:adr) (La: LList h a) : list nat :=
     match nullp a with
       left  p => nil
     | right p => value (access h a (LL_alloc_val La p))
                  ::LL_list (LL_next La p)
     end.
```

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

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

Linear Search. Doing the naive linear search follows the same scheme:

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

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

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

The proof goes 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 $\forall a\,(La : \texttt{LList}\,h\,a), P\ a\ La$, with the property to be proven depending on the proof of (`LList` h a). There is a special command to derive this more powerful principle:

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

Then the proof of the lemma starts with:

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

In this section, we give an overview of different ways to use automation in Coq including a brief overview of the tactic language `Ltac` and of proofs by reflection.

5.1 Existing Automated Tactics

We have already mentioned and used basic automated tactic in Coq. The `auto` and `trivial` tactics combine lemmas given by the user. Some tactics like `tauto`, `firstorder` and `intuition` deal with the propositional structure of propositions. There are generic tactics like `ring` and `field` to deal with algebraic manipulations in specific structures; the user can extend these tactics for new data-types and operations. A tactic like `omega` implements a decision procedure for linear arithmetic. There are also more specialized tactics: `gappa` and `interval` (G. Melquiond) deal with floating point numbers and real numbers (using computation on floating point numbers to establish results on real numbers); `Psatz` (F. Besson and E. Makarov) deals with arithmetic over ordered rings; `Nsatz` (L. Pottier) solves equalities in integral domains; `ergo` (S. Lescuyer) implements a small SMT solver. There is an integrated library implementing rewriting modulo Associativity-Commutativity developed by Th. Braibant and D. Pous. These tactics are fully integrated in Coq. Some other approaches use external tools as oracles producing traces. Early experiments were done with resolution (M. Bezem, D. Hendriks and H. de Nivelle). Several attempts are dealing

with rewriting tools: first with Elan (C. Alvarado, P. Crégut) and with CiME and the corresponding Coq library coccinelle (E. Contejean, X. Urbain et al.) and also the color library (F. Blanqui). As mentioned earlier, there is ongoing work on integrating SAT/SMT solvers : MiniSat, VeriT (B. Grégoire, C. Keller et al. [2]).

Some of the algorithms behind these tactics are quite involved. They are mainly developed at the Ocaml level. Coq has a `plugin` mechanism which allows to dynamically load new developments containing Ocaml code. So a new tactic can be developed independently of the system itself.

An important point is that the tactic will always provide a proof-term that is checked again by the Coq trusted kernel. So using third-party tactics will not lead to accept a false proposition as a theorem.

5.2 A Language for Writing Tactics

Writing powerful tactics like the ones mentioned before is a lot of work and requires to understand part of the internals of Coq. For many simple tasks, we can work at a higher level. The Ltac language designed by D. Delahaye [21] allows to write complex tactics without writing ML code. The language implements many useful schemes like (non-linear) pattern-matching on goals, generation of fresh names (`fresh` *name*), type-checking (`type of` *term*), and the programming capabilities of Coq can be used for integrating complex data-structures. Examples of pattern-matching constructs are:

```
match goal with
   id:?A /\ ?B |- ?A => destruct id; trivial
   | _ => idtac
end
match goal with |- context[?a+0] => rewrite ... end
```

These constructions are tactics that can be used anonymously everywhere in a proof. It is also possible to declare tactics with arguments like in the following examples related to our `board` example.

```
Coq < Ltac triple_force tac t :=
Coq <      let x := fresh "x" in
Coq <      let y := fresh "y" in
Coq <      let z := fresh "z" in
Coq <      destruct t as (x,y,z); tac x; tac y; tac z.
triple_force is defined
```

The tactic `triple_force` takes as argument a tactic `tac` and a term `t`. It will work when `t` has type `triple` M. It destructs t generating three new names for each component and applies sequentially the tactic `tac` to these three new objects. Remember a `board` is a triple of triples of colors. Given a term `t` of type `board`, the tactic `board_force` will generate a new color variable for each place in the board and apply `tac` to each of them.

```
Coq < Ltac board_force tac t :=
Coq < let tac1 := triple_force tac in triple_force tac1 t.
board_force is defined
```

These user defined tactics can now be used like other tactics.

5.3 Reflexive Tactics

An important rule of the theory of Coq is the convertibility rule:

$$\frac{\Gamma \vdash U : s \qquad \Gamma \vdash t : T \qquad T \equiv U}{\Gamma \vdash t : U}$$

It says that a proof t of a proposition T is also a proof of U when T and U are convertible (equal modulo computation). A consequence of that rule is that computation is part of proof verification (this is why termination is so important to keep decidability of type-checking). All languages with possible computations inside formulas integrates this sort of rule. A specificity of Coq is that, the set of terms integrates a powerful programming language (functional kernel of CAML), so heavy computation can occur during type-checking. A good news is that this capability can be really helpful for conducting proofs. The ssreflect tactic language [1] developed when proving the four-color theorem heavily relies on this possibility. More and more tactics are developed directly in Coq thanks to its computational power. In the meantime, it was necessary to improve the efficiency of reduction techniques inside the kernel of Coq. B. Grégoire introduced byte-code compiling, and more recently there has been work for integrating machine integers and primitive arrays [3].

Principle of Reflexive Tactics. We illustrate the principle of reflexive tactics on a typical example.

We want to prove a property P. We know of a (complex) mechanical way to prove P depending on the structure of P. One could use Ltac to combine existing tactics and build the proof but it can be inefficient and create very large proof terms.

So another idea is to try to program the tactic inside Coq itself. One cannot directly reason inside Coq on the structure of the proposition but one can introduce a new concrete type A (for instance an abstract syntax tree for the formula) and the only thing we need is an interpretation `r2P` from A to Prop plus a specific element $p : A$ such that (`r2P` p) is equivalent to the formula P we want to prove.

Now that we have a concrete type A, we can concentrate on implementing inside Coq whatever algorithm we like `r2b`, which should tell us for $p : A$ whether the corresponding proposition `r2P` p is provable.

So we expect the algorithm `r2b` to be of type $A \to$ `bool` and to satisfy the correctness property:

$$\texttt{rcor} : \forall p : A, \texttt{r2b}\, p = \texttt{true} \to \texttt{r2P}\, p$$

An algorithm which always answers `false` will be correct but useless. However no completeness is required.

Combining the different results, the term `rcor` p has type `r2b` p = `true` $\to P$ so we are left to prove that our complex program evaluates to `true`. The good news is that this proof is just a computation as soon as p is a closed term (without free variable). When we compute a closed term of type `bool` we can only get the values `true` or `false` as a result and we can now use the computation rule:

$$\frac{\texttt{eq\_refl true} : \texttt{true} = \texttt{true} \qquad \texttt{r2b}\, p \equiv \texttt{true}}{\texttt{eq\_refl true} : \texttt{r2b}\, p = \texttt{true}}$$

The proof term is completely trivial, all the proof checking is done by the computation as part of the equivalence-checking.

Finally, the term `rcor` p (`eq_refl true`) is well-typed of type P exactly when `r2b` $p \equiv$ `true`.

The first difficulty in implementing this method is that, we need to find for each formula P to be proven, a closed term $p : A$ such that `r2P` $p \equiv$ `true`. This is the reification part, usually done by ad-hoc tactics working with the proposition P. The object p can also reflect the trace of a proof of P done by an external tool. The term `r2b` p needs to reduce to a boolean value so p (or the part of p needed for computation) should be a closed term. The term p should preferably be small (otherwise the final proof term will be large). The main problems are that `r2b` needs to be proven correct and its reduction needs to be efficient.

This techniques is used in several Coq automated tactics like Ring, (R)Omega, Setoid Rewrite... and also in interfaces between Coq and other systems using traces.

Example of Reification. We give an example of reification techniques with an attempt to capture the structure of propositions (just `True` and `False` propositions and the conjunction). The abstract syntax tree will contain a `Var` n term for each sub-proposition that cannot be interpreted and an environment (represented here by a list of propositions), will keep track of the proposition associated with each variable.

```
Inductive form : Set :=
    T | F | Var : nat -> form
  | Conj : form -> form -> form.
Definition env := list Prop.
```

It is easy to define the interpretation of a formula (we first define the access function in the environment).

```
Fixpoint find_env (e:env) (n:nat) := match e with
    | nil         => True
    | cons a l => match n with | O => a
                               | S p => find_env l p
                  end
    end.
```

```
Fixpoint f2P e (f:form) {struct f} : Prop := match f with
    T => True | F => False | Var n => find_env e n
  | Conj p q => f2P e p /\ f2P e q
end.
```

We can now run an example:

```
Coq < Definition e := (True::False::(0=0)::nil).
Coq < Eval compute in
Coq <    (f2P e (Conj (Var 0) (Conj (Var 1) (Var 1)))).
      = True /\ 0 = 0 /\ False : Prop
```

We show a more involved use of Ltac to do the reification automatically. Ltac is first used to compute, from a Coq proposition, both an environment and the corresponding formula which introduces variables each time the proposition is not True, False or a conjunction.

```
Ltac env_form l f := match f with
   True => constr:(l,T)
 | False => constr:(l,F)
 | ?A /\ ?B => match env_form l A with (?l1,?A1) =>
                match env_form l1 B with (?l2,?A2) =>
                                constr:(l2,Conj A1 A2)
                end
               end
 | ?A => let n := eval compute in (length l)
             in constr:(cons A l,Var n)
 end.
```

Then the reification function computes the environment and the formula and applies the change tactic which will fail if the two expressions are not convertible.

```
Ltac reify := match goal with |- ?P =>
     match (env_form (nil (A:=Prop)) P) with (?l,?f) =>
     let e := eval compute in (rev l)
     in change (f2P e f) end
end.
```

Reification can be tested on a concrete example.

```
Coq < Lemma test1 : 0=0 /\ (False -> False) /\ 1=1 /\ (0=0).
Coq < reify.
1 subgoal
  ============================
   f2P ((0 = 0)::(False -> False)::(1 = 1)::(0 = 0)::nil)
     (Conj (Var 0) (Conj (Var 1) (Conj (Var 2) (Var 3))))
```

We see on this example that the two instances of 0=0 are not shared, a more sophisticated reification techniques can be implemented to avoid this behavior. Using reflexive techniques, one could design a simple tautology checker on the type form or extend the method to do trivial simplifications on the formula.

Application to the Board. Because a board has only a finite (but large) number of possible states, it is possible to design a reflexive tactic for deciding $\forall b : \texttt{board}, P(b) = \texttt{true}$. The relation `moves` between two boards can also be decided by looking at the appropriate invariant on the boards and this strategy can easily be implemented using a reflexive tactic.

6 Conclusion

In this tutorial, we have presented the basic constructions of the Coq proof assistant and we illustrated them on (small) examples of software verification taken from different classes of problems. We did not cover all aspects of Coq (modules, coinductive definitions, coercions, the ssreflect proof language are important notions not detailed here) and we just quickly mentioned other important features such as the tactic language, type classes... The interested user will need to use other material like the Coq reference manual [39] or other references mentioned in the introduction [9,15,37].

Our purpose was to illustrate some of the main features of Coq including inductive definitions, dependent types but also to give an overview of more elaborated libraries that can be reused and adapted to solve certain classes of problems. A proof assistant is like a programming environment: you can use it because you like the language primitives or because it offers a good set of pre-developed libraries (efficient, easy to reuse) that will make your programming task easier.

Why and When should I use Coq *?* Coq is *not* a direct tool to find bugs in C, Java or concurrent programs. But as a computer scientists specialized in formal methods and software verification, our job is mainly to design methods, tools to *help others* write correct programs!

Coq is *helpful* for developing complex mathematical proofs with high guaranty. This can be checking *theorems* in papers. During the last years, more and more communities of researchers have questioned their ability to check the correctness of proofs in academic papers. It is the case in the area of theory of programming languages (see the POPLmark Challenge [34]), computational cryptography [7], computer arithmetic [28] and certain "computational" mathematical proofs like the Kepler's conjecture (see the Flyspeck project [26]).

Coq can also be used as a *back-end* for program verification (when program correctness has been reduced to logical propositions) and also to develop and prove *pure functional* programs which can be abstract representations of algorithms or tools fully developed in a functional style.

Coq is not a tool that will magically *solve* your equation or your problem, but it is a general purpose language in which you can learn how to *program* your own solution. The entry cost is a lot *higher*, and it is not easy, but the possibilities are much *larger*. If you are lucky, somebody has developed a similar application that you can reuse.

Using a Theorem Prover as a Back-End. Some software verification techniques reduce program correctness to logical statements. Then the problem is to make sure these logical statements are correct.

One solution is to use an automatic first-order theorem prover or a SMT tool. These tools are automatic and very powerful. However, they may fail to prove a correct statement in which case you will have to better understand the algorithms in order to help the tool solve your problem. When they succeed, there is the question of could you trust or not the result ? There is always a possibility of a bug in the program (these tools involve complex algorithms and optimization methods) but even without bugs in the tool, there is the possibility that the theory you introduced for modeling your problem is inconsistent.

If instead you use a proof assistant based on a higher-order logic, the proof will be mainly interactive. It requires more expertise and time but there are few theoretical limitations on the extend of proofs you can do. Also it is possible to build models of the theory you are using that will avoid the pitfall of an inconsistent context, so the development in general will be much more reliable.

Where Automated Deduction Meets Proof Assistants. During the last years, there has been closer interactions between the two worlds: most first-order theorem provers produce traces that can be checked by an independent party. Proof assistants also provide automated strategies (either internal or external or with a combination of both). One question is: *do we have to choose ?*

A tool like the WHY3 PLATFORM [10] gives the possibility to describe problems in a high-level language (polymorphic multi-sorted logic including functions, algebraic data-types, axioms, lemma, modules) and to translate them efficiently to *multiple provers* including proof assistants, SMT/TPTP solvers, or specialized tools like GAPPA for floating-point arithmetic [20]. The WHY3 PLATFORM is used as a back-end for other environments for software verification like the Frama-C platform for Static analysis tools for C programs (CEA LIST & INRIA, B. Monate, L. Correnson [22]). Using this platform, you can submit your set of logical problems to many different provers and be happy if at least one of them solve each goal. If not, you can focus on the ones which are left and try to solve them interactively using a proof assistant.

A frequent question is also *Which proof assistant should I use?* There is no clear answer, using COQ or Isabelle/HOL for instance can be compared to the question using Ocaml or Haskell ? The choice may depend on *ideological* reasons: do you prefer classical or intuitionistic logic ? which trust base are you ready to accept ? do you need dependent types ? Often the choice will be based on more *practical* reasons: it is used in my team/company, an expert seats next door, I learned it at school, the library I need exists in that proof assistant...

What is true is that great achievements have been obtained by great people in all proof assistants Coq, HOL, PVS ... and that *biodiversity is healthy!*

References

1. Gonthier, G., Mahboubi, A., Tassi, E.: A small scale reflection extension for the Coq system. Technical report, Microsoft Research - Inria Joint Centre (MSR - INRIA) (2011), http://hal.inria.fr/inria-00258384
2. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 135–150. Springer, Heidelberg (2011)
3. Armand, M., Grégoire, B., Spiwack, A., Théry, L.: Extending Coq with Imperative Features and Its Application to SAT Verification. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 83–98. Springer, Heidelberg (2010)
4. Asperti, A., Coen, C.S., et al.: Matita, http://matita.cs.unibo.it/
5. Aspinall, D.: Proof general, http://proofgeneral.inf.ed.ac.uk/
6. Audebaud, P., Paulin-Mohring, C.: Proofs of randomized algorithms in Coq. Science of Computer Programming 74(8), 568–589 (2009)
7. Barthe, G., Grégoire, B., Béguelin, S.Z.: Formal certification of code-based cryptographic proofs. In: 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 90–101. ACM (2009), http://www.msr-inria.inria.fr/projects/sec/certicrypt
8. Bertot, Y.: Coq in a Hurry. Technical report, MARELLE - INRIA Sophia Antipolis (May 2010)
9. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer (2004), http://www.labri.fr/perso/casteran/CoqArt
10. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011)
11. Boldo, S., Clément, F., Filliâtre, J.-C., Mayero, M., Melquiond, G., Weis, P.: Formal Proof of a Wave Equation Resolution Scheme: The Method Error. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 147–162. Springer, Heidelberg (2010), http://hal.inria.fr/hal-00649240/PDF/RR-7826.pdf
12. Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in Coq. In: Antelo, E., Hough, D., Ienne, P. (eds.) Proceedings of the 20th IEEE Symposium on Computer Arithmetic, Tübingen, Germany, pp. 243–252 (2011), http://flocq.gforge.inria.fr/
13. Bolignano, D.: An approach to the formal verification of cryptographic protocols. In: CCS 1996 Proceedings of the 3rd ACM Conference on Computer and Communications Security (1996)
14. Chetali, B., Nguyen, Q.-H.: About the world-first smart card certificate with EAL7 formal assurances. In: Slides 9th ICCC, Jeju, Korea (September 2008), www.commoncriteriaportal.org/iccc/9iccc/pdf/B2404.pdf
15. Chlipala, A.: Certified Programming with Dependent Types. MIT Press (2011), http://adam.chlipala.net/cpdt/
16. Constable, R.L., Bates, J.L., Kreitz, C., van Renesse, R., et al.: Prl: Proof/program refinement logic, http://www.cs.cornell.edu/info/projects/nuprl/
17. Contejean, E., Paskevich, A., Urbain, X., Courtieu, P., Pons, O., Forest, J.: A3pat, an approach for certified automated termination proofs. In: Gallagher, J.P., Voigtländer, J. (eds.) Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010, pp. 63–72. ACM (2010)

18. Coquand, C., Coquand, T., Nurell, U., et al.: Agda, http://wiki.portal.chalmers.se/agda
19. Coquand, T., Huet, G.: Constructions: a Higher-order Proof System for Mechanizing Mathematics. In: Buchberger, B. (ed.) ISSAC 1985 and EUROCAL 1985. LNCS, vol. 203, pp. 151–184. Springer, Heidelberg (1985)
20. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. Transactions on Mathematical Software 37(1) (2010)
21. Delahaye, D.: A Proof Dedicated Meta-Language. In: Pfenning, F. (ed.) Logical Frameworks and Meta-Languages (LFM), Copenhagen (Denmark). Electronic Notes in Theoretical Computer Science (ENTCS), vol. 70(2), pp. 96–109. Elsevier (2002)
22. The Frama-C platform for static analysis of C programs, http://www.frama-c.cea.fr/
23. Geuvers, H., Wiedijk, F., Zwanenburg, J.: A Constructive Proof of the Fundamental Theorem of Algebra without Using the Rationals. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 96–111. Springer, Heidelberg (2002), http://www.cs.ru.nl/~freek/fta/
24. Gonthier, G.: Formal proof the four-color theorem. Notices of the AMS 55(11), 1382–1393 (2008), http://www.ams.org/notices/200811/tx081101382p.pdf
25. Gonthier, G.: Advances in the Formalization of the Odd Order Theorem. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, p. 2. Springer, Heidelberg (2011)
26. Hales, T.: Flyspeck project. The purpose of the flyspeck project is to produce a formal proof of the Kepler Conjecture, http://code.google.com/p/flyspeck/
27. Harrison, J.: The HOL Light theorem prover, http://www.cl.cam.ac.uk/~jrh13/hol-light/
28. Harrison, J.V.: A Machine-Checked Theory of Floating Point Arithmetic. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 113–130. Springer, Heidelberg (1999)
29. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009), http://compcert.inria.fr/
30. Letouzey, P.: Extraction in Coq: An Overview. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) CiE 2008. LNCS, vol. 5028, pp. 359–369. Springer, Heidelberg (2008)
31. McBride, C., et al.: Epigram 2: an experimental dependently typed functional programming language, http://www.e-pig.org/darcs/Pig09/web/
32. Morrisett, G., et al.: The Ynot project, http://ynot.cs.harvard.edu/
33. Norrish, M., Slind, K., et al.: HOL theorem-proving system (HOL4), http://hol.sourceforge.net/
34. University of Pennsylvania & University of Cambridge. The POPLmark challenge (2006), https://alliance.seas.upenn.edu/~plclub/cgi-bin/poplmark/
35. Owre, S., Rushby, J., Shankar, N., et al.: PVS specification and verification system, http://pvs.csl.sri.com/
36. Paulson, L., Nipkow, T., Wenzel, M.: Isabelle, http://www.cl.cam.ac.uk/research/hvg/isabelle
37. Pierce, B.C., Casinghino, C., Greenberg, M., Sjöberg, V., Yorgey, B.: Software Foundations. University of Pennsylvania (2011), http://www.cis.upenn.edu/~bcpierce/sf/

38. Sozeau, M.: Programing finger trees in Coq. In: Hinze, R., Ramsey, N. (eds.) Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, pp. 13–24. ACM (2007)
39. The Coq Development Team. The Coq Proof Assistant Reference Manual – Version V8.3 (2010), http://coq.inria.fr
40. Théry, L., Hanrot, G.: Primality Proving with Elliptic Curves. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 319–333. Springer, Heidelberg (2007), http://coqprime.gforge.inria.fr/