Logic
(Métodos Formais em Engenharia de Software)

Maria João Frade

HASLab - INESC TEC
Departmento de Informática, Universidade do Minho

2012/2013

Roadmap

- **Inductive Definitions**
  - inductive types and its elimination mechanisms;
  - proof by induction; case analysis; general recursion;
  - relations as inductive types; logical connectives as inductive types.
- **More about Coq**
  - the Coq library; searching the environment;
  - useful tactics and commands; combinnig tactics; automatic tactics.
- **Programming and Proving in Coq**
  - some datatypes of programming;
  - functional correctness; partiality; specification types;
  - program extraction;
  - non-structural recursion.

Inductive Definitions

Induction

Induction is a basic notion in logic and set theory.

- When a set is defined inductively we understand it as being "built up from the bottom" by a set of basic constructors.
- Elements of such a set can be decomposed in "smaller elements" in a well-founded manner.
- This gives us principles of
  - *"proof by induction"* and
  - *"function definition by recursion"*.

Inductive types

We can define a new type I inductively by giving its *constructors* together with their types which must be of the form

$$\boxed{\tau_1 \to \ldots \to \tau_n \to I \quad , \text{ with } \quad n \geq 0}$$

- Constructors (which are the *introduction rules* of the type I) give the canonical ways of constructing one element of the new type I.
- The type I defined is the smallest set (of objects) closed under its introduction rules.
- The inhabitants of type I are the objects that can be obtained by a finite number of applications of the type constructors.

Type I (under definition) can occur in any of the "domains" of its constructors. However, the occurrences of I in τ_i must be in *positive positions* in order to assure the well-foundedness of the datatype.

For instance, assuming that I does not occur in types A and B: $I \to B \to I$, $A \to (B \to I) \to I$ or $((I \to A) \to B) \to A \to I$ are valid types for a constructor of I, but $(I \to A) \to I$ or $((A \to I) \to B) \to A \to I$ are not.

Induction types - examples

The inductive type $\mathbb{N}$: Set of natural numbers has two constructors

$$0 : \mathbb{N}$$
$$\mathsf{S} : \mathbb{N} \to \mathbb{N}$$

A well-known example of a higher-order datatype is the type $\mathbb{O}$: Set of ordinal notations which has three constructors

$$\begin{array}{lcl} \text{Zero} & : & \mathbb{O} \\ \text{Succ} & : & \mathbb{O} \to \mathbb{O} \\ \text{Lim} & : & (\mathbb{N} \to \mathbb{O}) \to \mathbb{O} \end{array}$$

To program and reason about an inductive type we must have means to analyze its inhabitants.

The *elimination rules* for the inductive types express ways to use the objects of the inductive type in order to define objects of other types, and are associated to new computational rules.

Recursors

When an inductive type is defined in a type theory the theory should automatically generate a scheme for proof-by-induction and a scheme for primitive recursion.

- The inductive type comes equipped with a *recursor* that can be used to define functions and prove properties on that type.
- The recursor is a constant $\mathbf{R}_I$ that represents the structural induction principle for the elements of the inductive type I, and the computation rule associated to it defines a safe recursive scheme for programming.

For example, $\mathbf{R}_\mathbb{N}$, the recursor for $\mathbb{N}$, has the following typing rule:

$$\frac{\Gamma \vdash P : \mathbb{N} \to \mathsf{Type} \quad \Gamma \vdash a : P\,0 \quad \Gamma \vdash a' : \Pi x{:}\mathbb{N}.\, P\,x \to P\,(\mathsf{S}\,x)}{\Gamma \vdash \mathbf{R}_\mathbb{N}\,P\,a\,a' : \Pi n{:}\mathbb{N}.\, P\,n}$$

and its computation rules are

$$\begin{array}{lcl} \mathbf{R}_\mathbb{N}\,P\,a\,a'\,0 & \to & a \\ \mathbf{R}_\mathbb{N}\,P\,a\,a'\,(\mathsf{S}\,x) & \to & a'\,x\,(\mathbf{R}_\mathbb{N}\,P\,a\,a'\,x) \end{array}$$

Proof-by-induction scheme

The proof-by-induction scheme can be recovered by setting P to be of type $\mathbb{N} \to \mathsf{Prop}$.

Let $\mathsf{ind}_\mathbb{N} := \lambda P{:}\mathbb{N} \to \mathsf{Prop}.\, \mathbf{R}_\mathbb{N}\,P$ we obtain the following rule

$$\frac{\Gamma \vdash P : \mathbb{N} \to \mathsf{Prop} \quad \Gamma \vdash a : P\,0 \quad \Gamma \vdash a' : \Pi x{:}\mathbb{N}.\, P\,x \to P\,(\mathsf{S}\,x)}{\Gamma \vdash \mathsf{ind}_\mathbb{N}\,P\,a\,a' : \Pi n{:}\mathbb{N}.\, P\,n}$$

This is the well known structural induction principle over natural numbers. It allows to prove some universal property of natural numbers ($\forall n{:}\mathbb{N}.\, P n$) by induction on n.

Primitive recursion scheme

The primitive recursion scheme (allowing dependent types) can be recovered by setting $P : \mathbb{N} \to \mathsf{Set}$.

Let $\mathsf{rec}_{\mathbb{N}} \;:=\; \lambda P{:}\mathbb{N}\to\mathsf{Set}.\, \mathbf{R}_{\mathbb{N}}\, P$ we obtain the following rule

$$\frac{\Gamma \vdash T : \mathbb{N}\to\mathsf{Set} \quad \Gamma \vdash a : T\,0 \quad \Gamma \vdash a' : \Pi x{:}\mathbb{N}.\, T\,x \to T\,(\mathsf{S}\,x)}{\Gamma \vdash \mathsf{rec}_{\mathbb{N}}\, T\, a\, a' : \Pi n{:}\mathbb{N}.\, T\, n}$$

We can define functions using the recursors.

For instance, a function that doubles a natural number can be defined as follows:

$$\mathsf{double} \;:=\; \mathsf{rec}_{\mathbb{N}}\,(\lambda n{:}\mathbb{N}.\,\mathbb{N})\,0\,(\lambda x{:}\mathbb{N}.\,\lambda y{:}\mathbb{N}.\,\mathsf{S}\,(\mathsf{S}\,y))$$

This approach gives safe way to express recursion without introducing non-normalizable objects.

However, codifying recursive functions in terms of elimination constants is quite far from the way we are used to program. Instead we usually use general recursion and case analysis.

Case analysis

Case analyses gives an elimination rule for inductive types.

For instance, $n : \mathbb{N}$ means that n was introduced using either 0 or S, so we may define an object match n with $\{0 \Rightarrow b_1 \mid (\mathsf{S}x) \Rightarrow b_2\}$ in another type σ depending on which constructor was used to introduce n.

A typing rule for this construction is

$$\frac{\Gamma \vdash n : \mathbb{N} \quad \Gamma \vdash b_1 : \sigma \quad \Gamma, x : \mathbb{N} \vdash b_2 : \sigma}{\Gamma \vdash \mathsf{match}\ n\ \mathsf{with}\ \{0 \Rightarrow b_1 \mid (\mathsf{S}\,x) \Rightarrow b_2\} : \sigma}$$

and the associated computation rules are

$$\begin{array}{lcl} \mathsf{match}\ 0\ \mathsf{with}\ \{0 \Rightarrow b_1 \mid \mathsf{S} \Rightarrow b_2\} & \to & b_1 \\ \mathsf{match}\ (\mathsf{S}\,e)\ \mathsf{with}\ \{0 \Rightarrow b_1 \mid (\mathsf{S}\,x) \Rightarrow b_2\} & \to & b_2\,[e/x] \end{array}$$

The case analysis rule is very useful but it does not give a mechanism to define recursive functions.

General recursion

Functional programming languages feature *general recursion*, allowing recursive functions to be defined by means of pattern-matching and a general fixpoint operator to encode recursive calls.

The typing rule for $\mathbb{N}$ fixpoint expressions is

$$\frac{\Gamma \vdash \mathbb{N}\to\theta : s \quad \Gamma, f : \mathbb{N}\to\theta \vdash e : \mathbb{N}\to\theta}{\Gamma \vdash (\mathsf{fix}\ f = e) : \mathbb{N}\to\theta}$$

and the associated computation rules are

$$\begin{array}{lcl} (\mathsf{fix}\ f = e)\,0 & \to & e[(\mathsf{fix}\ f = e)/f]\,0 \\ (\mathsf{fix}\ f = e)\,(\mathsf{S}\,x) & \to & e[(\mathsf{fix}\ f = e)/f]\,(\mathsf{S}\,x) \end{array}$$

Of course, this approach opens the door to the introduction of non-normalizable objects.

Using this, the function that doubles a natural number can be defined by

$$(\mathsf{fix}\ \mathsf{double} = \lambda n : \mathbb{N}.\, \mathsf{match}\ n\ \mathsf{with}\ \{0 \Rightarrow 0 \mid (\mathsf{S}\,x) \Rightarrow \mathsf{S}\,(\mathsf{S}\,(\mathsf{double}\,x))\})$$

About termination

- Checking convertibility between types may require computing with recursive functions. So, the combination of non-normalization with dependent types leads to undecidable type checking.
- To enforce decidability of type checking, proof assistants either require recursive functions to be encoded in terms of recursors or allow restricted forms of fixpoint expressions.
- A usual way to ensure termination of fixpoint expressions is to impose syntactical restrictions constraining all recursive calls to be applied to terms structurally smaller than the formal argument of the function.
- Another way to ensure termination is to accept a measure function that specifies how the argument "decreases" between recursive function calls.

Computation

Recall that typing judgments in Coq are of the form $E\,|\,\Gamma \vdash M : A$, where E is the global environment and Γ is the local context.

Computations are performed as series of *reductions*.

β-reduction for compute the value of a function for an argument:

$$(\lambda x\!:\!A.\,M)\,N \quad \to_\beta \quad M[N/x]$$

δ-reduction for unfolding definitions:

$$M \quad \to_\delta \quad N \qquad \text{if } (M := N) \in E\,|\,\Gamma$$

ι-reduction for primitive recursion rules, general recursion and case analysis

ζ-reduction for local definitions: let $x := N$ in $M \quad \to_\zeta \quad M[N/x]$

Note that the conversion rule is

$$\frac{E\,|\,\Gamma \vdash M : A \qquad E\,|\,\Gamma \vdash B : s}{E\,|\,\Gamma \vdash M : B} \quad \text{if } A =_{\beta\iota\delta\zeta} B \text{ and } s \in \{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}\}$$

Natural numbers

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

The declaration of this inductive type introduces in the global environment not only the constructors O and S but also the recursors: `nat_rect`, `nat_ind` and `nat_rec`

```
Check nat_rect.
```

```
nat_rect
     : forall P : nat -> Type,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

```
Print nat_ind.
```

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

```
Print nat_rec.
```

```
nat_rec = fun P : nat -> Set => nat_rect P
     : forall P : nat -> Set,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```


Vectors of length n over A.

```
Inductive vector (A : Type) : nat -> Type :=
  | Vnil : vector A 0
  | Vcons : A -> forall n : nat, vector A n -> vector A (S n).
```

Remark the difference between the two parameters A and n:

- A is a general parameter, global to all the introduction rules,
- n is an index, which is instantiated differently in the introduction rules.

The type of constructor `Vcons` is a dependent function.

```
Variables b1 b2 :  B.
Check (Vcons _ b1 _ (Vcons _ b2 _ (Vnil _))).
```

```
Vcons B b1 1 (Vcons B b2 0 (Vnil B))  : vector B 2
```

```
Check vector_rect.
```

```
vector_rect
     : forall (A : Type) (P : forall n : nat, vector A n -> Type),
       P 0 (Vnil A) ->
       (forall (a : A) (n : nat) (v : vector A n),
        P n v -> P (S n) (Vcons A a n v)) ->
       forall (n : nat) (v : vector A n), P n v
```


Equality

In Coq, the propositional equality between two inhabitants a and b of the same type A, noted $a = b$, is introduced as a family of recursive predicates "to be equal to a", parameterized by both a and its type A. This family of types has only one introduction rule, which corresponds to reflexivity.

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  | refl_equal : (eq A x x).
```

The induction principle of eq is very close to the Leibniz's equality but not exactly the same.

```
Check eq_ind.
```

```
eq_ind : forall (A : Type) (x : A) (P : A -> Prop),
         P x -> forall y : A, x = y -> P y
```

Notice that the syntax "`a = b`" is an abbreviation for "`eq a b`", and that the parameter A is implicit, as it can be inferred from a.

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  | refl_equal : x = x.
```

Relations as inductive types

Some relations can also be introduced as an inductive family of propositions. For instance, the order $n \leq m$ on natural numbers is defined as follows in the standard library:

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

- Notice that in this definition `n` is a general parameter, while the second argument of `le` is an index. This definition introduces the binary relation $n \leq m$ as the family of unary predicates "to be greater or equal than a given n", parameterized by n.
- The Coq system provides a syntactic convention, so that "`le x y`" can be written "`x <= y`".
- The introduction rules of this type can be seen as rules for proving that a given integer n is less or equal than another one. In fact, an object of type $n \leq m$ is nothing but a proof built up using the constructors `le_n` and `le_S`.

Logical connectives in Coq

In the Coq system, most logical connectives are represented as inductive types, except for $\Rightarrow$ and $\forall$ which are directly represented by $\rightarrow$ and Π-types, negation which is defined as the implication of the absurd and equivalence which is defined as the conjunction of two implications.

```
Definition not := fun A : Prop => A -> False.

Notation "~ A" := (not A) (at level 75, right associativity).
```

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

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

```
Inductive and (A : Prop) (B : Prop) : Prop :=
  | conj : A -> B -> (and A B).

Notation "A /\ B" := (and A B) (at level 80, right associativity).
```


Logical connectives in Coq

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

Notation "A \/ B" := (or A B) (at level 85, right associativity).
```

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

`exists x:A, P` is an abbreviation of `ex A (fun x:A => P)`.

```
Definition iff (P Q:Prop) := (P -> Q) /\ (Q -> P).

Notation "P <-> Q" := (iff P Q) (at level 95, no associativity).
```

The constructors are the introduction rules.
The induction principle gives the elimination rules.
All the (constructive) logical rules are now derivable.

More About Coq

The Coq library

Proof development often take advantage from the large base of definitions and facts found in the Coq library.

- *The initial library*: it contains elementary logical notions and datatypes. It constitutes the basic state of the system directly available when running Coq.
- *The standard library*: general-purpose libraries containing various developments of Coq axiomatizations about sets, lists, sorting, arithmetic, etc. This library comes with the system and its modules are directly accessible through the `Require` command.
- *Users' contributions*: user-provided libraries or developments are provided by Coq users' community. These libraries and developments are available for download.

Coq standard library

In the Coq system most usual datatypes are represented as inductive types and packages provide a variety of properties, functions, and theorems around these datatypes.

Some often used packages:

`Logic`	Classical logic and dependent equality
`Arith`	Basic Peano arithmetic
`ZArith`	Basic relative integer arithmetic
`Bool`	Booleans (basic functions and results)
`Lists`	Polymorphic lists and Streams
`Sets`	Sets (classical, constructive, finite, infinite, power set, etc.)
`FSets`	Specification and implementations of finite sets and finite maps
`QArith`	Axiomatization of rational numbers
`Reals`	Formalization of real numbers
`Relations`	Relations (definitions and basic results)
...	

Searching the environment

Some useful commands to find already existing proofs of facts in the environment.

- `Search` *ident* - displays the name and type of all theorems of the current context whose statement's conclusion has the form (*ident* t1 .. tn)
- `SearchAbout` *ident* - displays the name and type of all objects (theorems, axioms, etc) of the current context whose statement contains *ident*.
- `SearchPattern` *pattern* - displays the name and type of all theorems of the current context which matches the expression *pattern*.
- `SearchRewrite` *pattern* - displays the name and type of all theorems of the current context whose statement's conclusion is an equality of which one side matches the expression *pattern*.

Check the following commands:

```
Search le.
SearchAbout le.
SearchPattern (le (_ + _) (_ + _)).
SearchPattern (_ + _ <= _ + _).
SearchRewrite (_ + (_ - _)).
```


Basic tactics

- `intro`, `intros` – introduction rule for Π (several times)
- `apply` – elimination rule for Π
- `assumption` – match conclusion with an hypothesis
- `exact` – gives directly the exact proof term of the goal

Tactics for first-order reasoning

Proposition (P)	Introduction	Elimination (H of type P)
$\bot$		elim H, contradiction
$\neg A$	intro	apply H
$A \wedge B$	split	elim H, destruct H as [H1 H2]
$A \Rightarrow B$	intro	apply H
$A \vee B$	left, right	elim H, destruct H as [H1|H2]
$\forall x : A. Q$	intro	apply H
$\exists x : A. Q$	exists *witness*	elim H, destruct H as [x H1]

Tactics for equational reasoning

- `rewrite` – rewrites a goal using an equality.
- `rewrite <-` – rewrites a goal using an equality in the reverse direction.
- `reflexivity` – reflexivity property for equality.
- `symmetry` – symmetry property for equality.
- `transitivity` – transitivity property for equality.
- `replace a with b` – replaces `a` by `b` while generating the subgoal `a=b`.
- ...

Convertibility tactics

- `simpl`, `red`, `cbv`, `lazy`, `compute` – performs evaluation.
- `unfold` – applies the δ rule for a transparent constant.
- `pattern` – performs a beta-expansion on the goal.
- `change` – replaces the goal by a convertible one.
- ...

Tactics for inductive reasoning

- `elim` – to apply the corresponding induction principle.
- `induction` – performs induction on an identifier.
- `case`, `destruct` – performs case analysis.
- `constructor` – applies to a goal such that the head of its conclusion is an inductive constant.
- `discriminate` – discriminates objects built from different constructors.
- `injection` – applies the fact that constructors of inductive types are injections.
- `inversion` – given an inductive type instance, find all the necessary condition that must hold on the arguments of its constructors
- ...

Other useful tactics and commands

- `clear` – removes an hypothesis from the environment.
- `generalize` – reintroduces an hypothesis into the goal.
- `cut`, `assert` – proves the goal through an intermediate result.
- `absurd` – applies `False` elimination.
- `contradict` – allows to manipulate negated hypothesis and goals.
- `refine` – allows to give an exact proof but still with some holes ("_").
- ...

- `Admitted` – aborts the current proof and replaces the statement by an axiom that can be used in later proofs.
- `Abort` – aborts the current proof without saving anything.

Combining tactics

The basic tactics can be combined into more powerful tactics using tactics combinators, also called *tacticals*.

- `t1 ; t2` – applies tactic `t1` to the current goal and then `t2` to each generated subgoal.
- `t1 || t2` – applies tactic `t1`; if it fails then applies `t2`.
- `t ; [ t1 | ...| tn]` – applies `t` and then `ti` to the i-th generated subgoals; there must be exactly n subgoals generated by `t`.
- `idtac` – does nothing
- `try t` – applies `t` if it does not fail; otherwise does nothing.
- `repeat t` – repeats `t` as long as it does not fail.
- `solve t` – applies `t` only if it solves the current goal.
- ...

Automatic tactics

- `trivial` – tries those tactics that can solve the goal in one step.
- `auto` – tries a combination of tactics `intro`, `apply` and `assumption` using the theorems stored in a database as hints for this tactic.
- `eauto` – like `auto` but more powerful but also more time-consuming.
- `autorewrite` – repeats rewriting with a collection of theorems, using these theorems always in the same direction.
- `tauto` – useful to prove facts that are tautologies in intuitionistic PL.
- `intuition` – useful to prove facts that are tautologies in intuitionistic PL.
- `firstorder` – useful to prove facts that are tautologies in intuitionistic FOL.
- `ring` – does proves of equality for expressions containing addition and multiplication.
- `omega` – proves systems of linear inequations (sums of $n * x$ terms).
- `field` – like `ring` but for a field structure (it also considers division).
- `fourier` – like `omega` but for real numbers.
- `subst` – replaces all the occurrences of a variable defined in the hypotheses.

Controlling automation

Several *hint databases* are defined in the Coq standard library. The actual content of a database is the collection of the hints declared to belong to this database in each of the various modules currently loaded.

- `Hint Resolve` – add theorems to the database of hints to be used by `auto` using `apply`.
- `Hint Rewrite` – add theorems to the database of hints to be used by `autorewrite`
- ...

Defined databases: `core`, `arith`, `zarith`, `bool`, `datatypes`, `sets`, `typeclass_instances`, `v62`.

One can optionally declare a hint database using the command `Create HintDb`. If a hint is added to an unknown database, it will be automatically created.

Exercises

Load the file lessonCoq2.v in the Coq proof assistant. Analyse the examples and solve the exercises proposed.

Programming and Proving in Coq

Some datatypes of programming

```
Inductive unit : Set :=  tt : unit.

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

Inductive nat : Set :=  O : nat | S : nat -> nat.

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

Inductive identity (A : Type) (a : A) : A -> Type :=
    refl_identity : identity A a a.
```

Some operations on bool are also provided: andb (with infix notation &&), orb (with infix notation ||), xorb, implb and negb.

Some datatypes of programming

```
Inductive sum (A B : Type) : Type :=  inl : A -> A + B
                                    | inr : B -> A + B.

Inductive prod (A B : Type) : Type :=  pair : A -> B -> A * B.

Definition fst (A B : Type) (p : A * B) := let (x, _) := p in x.

Definition snd (A B : Type) (p : A * B) := let (_, y) := p in y.
```

The constructive sum {A}+{B} of two propositions A and B.

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

If-then-else

- The sumbool type can be used to define an "if-then-else" construct in Coq.
- Coq accepts the syntax `if` *test* `then` ... `else` ... when *test* has either of type `bool` or `{A}+{B}`, with propositions `A` and `B`.
- Its meaning is the pattern-matching

```
match test with
 | left H => ...
 | right H => ...
end.
```

- We can identify `{P}+{~P}` as the type of decidable predicates:

The standard library defines many useful predicates, e.g.

```
le_lt_dec : forall n m : nat, {n <= m} + {m < n}
Z_eq_dec : forall x y : Z, {x = y} + {x <> y}
Z_lt_ge_dec : forall x y : Z, {x < y} + {x >= y}
```


If-then-else

A function that checks if an element is in a list.

```
Fixpoint elem (a:Z) (l:list Z) {struct l} : bool :=
    match l with
      | nil => false
      | cons x xs => if Z_eq_dec x a then true else (elem a xs)
    end.
```

Exercise:

Prove that

```
Theorem elem_corr : forall (a:Z) (l1 l2:list Z),
               elem a (app l1 l2) = orb (elem a l1) (elem a l2).
```


The "subset" type

- Coq's type system allows to combine a datatype and a predicate over this type, creating "the type of data that satisfies the predicate". Intuitively, the type one obtains represents a subset of the initial type.

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

- Given `A:Type` and `P:A->Prop`, the syntactical convention for `(Sig A P)` is the construct `{x:A | P x}`. (Predicate `P` is the *caracteristic function* of this set).
- We may build elements of this set as `(exist x p)` whenever we have a witness `x:A` with its justification `p:(P x)`.
- From such a `(exist x p)` we may in turn *extract* its witness `x:A` .
- In technical terms, one says that `sig` is a "*dependent sum*" or a *Σ-type*.

The "subset" type

A value of type `{x:A | P x}` should contain a *computation component* that says how to obtain a value v and a *certificate*, a proof that v satisfies predicate `P`.

A variant `sig2` with two predicates is also provided.

```
Inductive sig2 (A : Type) (P Q : A -> Prop) : Type :=
    exist2 : forall x : A, P x -> Q x -> sig2 A P Q
```

The notation for `(sig2 A P Q)` is `{x:A | P x & Q x}`.

Functional correctness

There are two approaches to defining functions and providing proofs that they satisfy a given specification:

- To define these functions with a *weak specification* and then add *companion lemmas*.
  For instance, we define a function $f : A \to B$ and we prove a statement of the form $\forall x\!:\!A, R\,x\,(f\,x)$, where R is a relation coding the intended input/output behaviour of the function.
- To give a *strong specification* of the function: the type of this function directly states that the input is a value x of type A and that the output is the combination of a value v of type B and a proof that v satisfies $R\,x\,v$.
  This kind of specification usually relies on dependent types.

Partiality

The Coq system does not allow the definition of partial functions (i.e. functions that give a run-time error on certain inputs). However we can enrich the function domain with a precondition that assures that invalid inputs are excluded.

- A partial function from type A to type B can be described with a type of the form $\forall x\!:\!A, P\,x \to B$, where P is a predicate that describes the function's domain.
- Applying a function of this type requires two arguments: a term t of type A and a proof of the precondition $P\,t$.

Example: the function head

An attempt to define the head function as follows will fail!

```
Definition head (A:Type) (l:list A) : A :=
  match l with
  | cons x xs => x
  end.
Error: Non exhaustive pattern-matching: no clause found
       for pattern nil
```

To overcome the above difficulty, we need to:

- consider a precondition that excludes all the erroneous argument values;
- pass to the function an additional argument: a proof that the precondition holds;
- the match constructor return type is lifted to a function from a proof of the precondition to the result type.
- any invalid branch in the match constructor leads to a logical contradiction (it violates the precondition).

Example: the function head

```
Definition head (A:Type) (l:list A) : l<>nil -> A.
refine (
  match l as l' return l'<>nil -> A with
  | nil => fun H => _
  | cons x xs => fun H => x
  end ).
elimtype False; apply H; reflexivity.
Defined.
```

```
Print Implicit head.
head : forall (A : Type) (l : list A), l <> nil -> A

Arguments A, l are implicit
```

Example: the function head

The specification of head is:

```
Definition headPre (A:Type) (l:list A) : Prop := l<>nil.

Inductive headRel (A:Type) (x:A) : list A -> Prop :=
    headIntro : forall l, headRel x (cons x l).
```

The correctness of function head is thus given by the following theorem:

```
Lemma head_correct : forall (A:Type) (l:list A) (p:headPre l),
                     headRel (head  p) l.
Proof.
induction l.
intro H; elim H; reflexivity.
intros; destruct l; [simpl; constructor | simpl; constructor].
Qed.
```


Extraction

- Conventional programing languages do not provide dependent types and well-typed functions in Coq do not always correspond to well-typed functions in the target programing language.
- In CIC functions may contain subterms corresponding to proofs that have practically no interest with respect to the final value.
- The computations done in the proofs correspond to verifications that should be done once and for all at compile-time, while the computation on the actual data needs to be done for each value presented to functions at run-time.
- Coq implements this mechanism of filtering the computational content from the objects - the so called extraction mechanism.
- The distinction between the sorts Prop and Set is used to mark the logical aspects that should be discharged during extraction or the computational aspects that should be kept.

Extraction

Coq supports different target languages: Ocaml, Haskell, Scheme.

```
Check head.
head : forall (A : Type) (l : list A), l <> nil -> A
```

```
Extraction Language Haskell.
Extraction Inline False_rect.
Extraction head.
```

```
head :: (List a1) -> a1
head l =
  case l of
    Nil -> Prelude.error "absurd case"
    Cons x xs -> x
```


Specification types

Using Σ-types we can express specification constrains in the type of a function - we simply restrict the codomain type to those values satisfying the specification.

- Consider the following definition of the inductive relation "x is the last element of list l", and the theorem specifing the function that gives the last element of a list.

```
Inductive Last (A:Type) (x:A) : list A -> Prop :=
| last_base : Last x (x :: nil)
| last_step : forall l y, Last x l -> Last x (y :: l).
```

```
Theorem last_correct : forall (A:Type) (l:list A),
                                    l<>nil -> { x:A | Last x l }.
```

- By proving this theorem we build an inhabitant of this type, and then we can extract the computational content of this proof, and obtain a function that satisfies the specification.
- The Coq system thus provides a certified software production tool, since the extracted programs satisfy the specifications described in the formal developments.

Specification types

Let us build an inhabitant of that type

```
Theorem last_correct : forall (A:Type) (l:list A),
                              l<>nil -> { x:A | Last x l }.
Proof.
induction l.
intro H; elim H; reflexivity.
intros. destruct l.
exists a; auto.
constructor.
elim IHl.
intros; exists x.
constructor. assumption.
discriminate.
Qed.
```


Program extraction

We can extract the computational content of the proof of the last theorem.

```
Extraction Language Haskell.

Extraction Inline False_rect.
Extraction Inline sig_rect.
Extraction Inline list_rect.

Extraction last_correct.
```

```
last_correct :: (List a1) -> a1
last_correct l =
  case l of
    Nil -> Prelude.error "absurd case"
    Cons a l0 -> (case l0 of
                    Nil -> a
                    Cons a0 l1 -> last_correct l0)
```


Case study: sorting a list

A simple characterisation of sorted lists consists in requiring that two consecutive elements be compatible with the $\leq$ relation.

We can codify this with the following predicate:

```
Open Scope Z_scope.

Inductive Sorted : list Z -> Prop :=
  | sorted0 : Sorted nil
  | sorted1 : forall z:Z, Sorted (z :: nil)
  | sorted2 : forall (z1 z2:Z) (l:list Z),
        z1 <= z2 -> Sorted (z2 :: l) -> Sorted (z1 :: z2 :: l).
```


Case study: sorting a list

To capture permutations, instead of an inductive definition we will define the relation using an auxiliary function that count the number of occurrences of elements:

```
Fixpoint count (z:Z) (l:list Z) {struct l} : nat :=
  match l with
  | nil => 0%nat
  | (z' :: l') =>
      match Z_eq_dec z z' with
      | left _ => S (count z l')
      | right _ => count z l'
      end
  end.
```

A list is a permutation of another when contains exactly the same number of occurrences (for each possible element):

```
Definition Perm (l1 l2:list Z) : Prop :=
                 forall z, count z l1 = count z l2.
```

Case study: sorting a list

Exercise:

Prove that Perm is an equivalence relation:

```
Lemma Perm_reflex : forall l:list Z, Perm l l.
Lemma Perm_sym : forall l1 l2, Perm l1 l2 -> Perm l2 l1.
Lemma Perm_trans : forall l1 l2 l3,
                 Perm l1 l2 -> Perm l2 l3 -> Perm l1 l3.
```

Exercise:

Prove the following lemmas:

```
Lemma Perm_cons : forall a l1 l2,
                           Perm l1 l2 -> Perm (a::l1) (a::l2).
Lemma Perm_cons_cons : forall x y l, Perm (x::y::l) (y::x::l).
```


Case study: sorting a list

A simple strategy to sort a list consist in iterate an "insert" function that inserts an element in a sorted list.

```
Fixpoint insert (x:Z) (l:list Z) {struct l} : list Z :=
  match l with
    nil => cons x nil
  | cons h t =>
        match Z_lt_ge_dec x h with
          left _ => cons x (cons h t)
        | right _ => cons h (insert x t)
        end
  end.
```

```
Fixpoint isort (l:list Z) : list Z :=
  match l with
    nil => nil
  | cons h t => insert h (isort t)
  end.
```


Case study: sorting a list

The theorem we want to prove is:

```
Theorem isort_correct : forall (l l':list Z),
                      l'=isort l -> Perm l l' /\ Sorted l'.
```

We will certainly need auxiliary lemmas... Let us make a prospective proof attempt:

```
Theorem isort_correct : forall (l l':list Z),
                      l'=isort l -> Perm l l' /\ Sorted l'.
induction l; intros.
unfold Perm; rewrite H; split; auto.
simpl. constructor. simpl in H.
rewrite H. (* ??????????? *)
```

```
  a : Z
  l : list Z
  IHl : forall l' : list Z, l' = isort l -> Perm l l' /\ Sorted l'
  l' : list Z
  H : l' = insert a (isort l)
  ============================
   Perm (a :: l) (insert a (isort l)) /\ Sorted (insert a (isort l))
```


Case study: sorting a list

It is now clear what are the needed lemmas:

```
Lemma insert_Perm :  forall x l, Perm (x::l) (insert x l).
unfold Perm; induction l.
simpl. reflexivity.
simpl insert. elim (Z_lt_ge_dec x a).  reflexivity.
intros. rewrite Perm_cons_cons.
pattern (x::l). simpl count. elim (Z_eq_dec z a).
intros. rewrite IHl; reflexivity.
intros. apply IHl.
Qed.
```

```
Lemma insert_Sorted :  forall x l, Sorted l -> Sorted (insert x l).
intros x l H; elim H; simpl.
constructor.
intro z; elim (Z_lt_ge_dec x z); intros.
constructor.
auto with zarith.
...
Qed.
```

Case study: sorting a list

Now we can conclude the proof of correctness...

```
Theorem isort_correct : forall (l l':list Z),
                        l'=isort l -> Perm l l' /\ Sorted l'.
Proof.
induction l; intros.
unfold Perm; rewrite H; split; auto.
simpl. constructor. simpl in H.
rewrite H. (* ?????????? *)
elim (IHl (isort l)); intros; split.
apply Perm_trans with (a::isort l).
unfold Perm. intro z. simpl. elim (Z_eq_dec z a). intros.
elim H0; reflexivity.
intros. elim H0. reflexivity.
apply insert_Perm.
apply insert_Sorted.
assumption.
Qed.
```


Case study: sorting a list

Exercise:

Complete the following proof and extract its computational content to an Haskell function.

```
Definition inssort : forall (l:list Z),
                              { l' | Perm l l' & Sorted l' }.
induction l.
  ...
Defined.
```


Non-structural recursion

When the recursion pattern of a function is not structural in the arguments, we are no longer able to directly use the derived recursors to define it.

Consider the Euclidean division algorithm written in Haskell

```
div :: Int -> Int -> (Int,Int)
div n d | n < d = (0,n)
        | otherwise = let (q,r) = div (n-d) d
                      in (q+1,r)
```

- In recent versions of Coq (after v8.1), a new command `Function` allows to directly encode general recursive functions.
- The `Function` command accepts a measure function that specifies how the argument "decreases" between recursive function calls.
- It generates proof-obligations that must be checked to guaranty the termination.

Non-structural recursion

```
Close Scope Z_scope.

Function div (p:nat*nat) {measure fst} : nat*nat :=
  match p with
  | (_,0) => (0,0)
  | (a,b) => if le_lt_dec b a
             then let (x,y) := div (a-b,b) in (1+x,y)
             else (0,a)
  end.
Proof.
intros.
simpl.
omega.
Qed.
```

The `Function` command generates a lot of auxiliary results related to the defined function. Some of them are powerful tools to reason about it.

Non-structural recursion

The `Function` command is also useful to provide "natural encodings" of functions that otherwise would need to be expressed in a contrived manner.

Exercise:

Complete the definition of the function merge, presenting a proof of its termination.

```
Function merge (p:list Z*list Z)
{measure (fun p=>(length (fst p))+(length (snd p)))} : list Z :=
  match p with
  | (nil,l) => l
  | (l,nil) => l
  | (x::xs,y::ys) => if Z_lt_ge_dec x y
                     then x::(merge (xs,y::ys))
                     else y::(merge (x::xs,ys))
  end.
```


Another example of correctness

A specification of the Euclidean division algorithm:

```
Definition divRel (args:nat*nat) (res:nat*nat) : Prop :=
          let (n,d):=args in let (q,r):=res in q*d+r=n /\ r<d.

Definition divPre (args:nat*nat) : Prop := (snd args)<>0.
```

A proof of correctness:

```
Theorem div_correct : forall (p:nat*nat),  divPre p -> divRel p (div p).
Proof.
unfold divPre, divRel.
intro p.
(* we make use of the specialised induction principle to conduct the proof... *)
functional induction (div p); simpl.
intro H; elim H; reflexivity.
(* a first trick: we expand (div (a-b,b)) in order to get rid of the let (q,r)=... *)
replace (div (a-b,b)) with (fst (div (a-b,b)),snd (div (a-b,b))) in IHp0.
simpl in *.
intro H; elim (IHp0 H); intros.
split.
(* again a similar trick: we expand "x" and "y0" in order to use an hypothesis *)
change (b + (fst (x,y0)) * b + (snd (x,y0)) = a).
rewrite <- e1.
omega.
(* and again... *)
change (snd (x,y0)<b); rewrite <- e1; assumption.
symmetry; apply surjective_pairing.
auto.
Qed.
```