MPRI 2-7-2: Proof Assistants

Matthieu Sozeau

January 15th 2023
Recap

Simple inductive types (datatypes):

Inductive bool := true | false.
Inductive list (A:Type) : Type :=
  nil | cons (hd:A) (tl:list A).
Inductive tree (A:Type) :=
  leaf | node (_:A) (_:nat->tree A).

Smallest type closed by introduction rules (constructors)

Parameters: cons : forall A:Type, A -> list A -> list A
Coq prelude: cons 0 nil : list nat
Recap: Elimination rules

Generated elimination scheme (not primitive):

```plaintext
nat_rect
  : forall P:nat->Type,
    P O -> (forall n, P n -> P (S n)) ->
    forall n, P n.
:= fun P h0 hS => fix F n :
    match n return P n with
    | O => h0
    | S k => hS k (F k)
end
```

Eliminator of recursive type =
deeped dependent pattern-matching + guarded fixpoint
Recap: Logical connectives

Logical connectives and their non-dependent elimination schemes:

**Inductive True : Prop := I.**
  True_rect : forall P:Type, P \rightarrow True \rightarrow P.

**Inductive False : Prop := .**
  False_rect : forall P:Type, False \rightarrow P

**Inductive and (A B:Prop) : Prop :=**
  conj (_,:A) (_,:B).
  and_rect : forall (A B:Prop) (P:Type), (A\rightarrow B\rightarrow P) \rightarrow A/\B \rightarrow P

**Inductive or (A B:Prop) : Prop :=**
  or_introl (_,:A) | or_intror (_,:B).
  or_ind : forall (A B P:Prop), (A \rightarrow P) \rightarrow (B \rightarrow P) \rightarrow P.
Overview

1. Inductive types
   - Equality
   - Arithmetic
   - Vectors

2. Theory of Inductive types
   - Strict Positivity
   - Dependent pattern-matching
   - Guarded fixpoint
   - The guard condition
Equality as an inductive family

\begin{align*}
\text{Inductive } & \quad \text{eq } (A : \text{Type}) \ (x : A) : A \to \text{Prop} := \\
& \quad \mid \text{eq\_refl } : \text{eq } A \ x \ x.
\end{align*}

**Elimination:**

- \text{eq\_rect: } \forall \ A \ x \ (P : A \to \text{Type}), \ P \ x \to \forall \ y, \ x=y \to P \ y

- **Dependent version (generated by Scheme):**

\begin{align*}
\forall \ A \ x \ (P : \forall \ z, \ x=z \to \text{Type}), \ P \ x \ \text{eq\_refl} \to \\
\forall \ y \ (e : x=y) \to P \ y \ e
\end{align*}
Dependent elimination needed to prove minimality:

\[
\text{match } n \ \text{return} \ n=0 \ \text{\(\lor\)} \ \exists \ m, \ n=S \ m \ \text{with} \\
\mid \ 0 \ \Rightarrow \ \text{inl} \ \text{eq_refl} : (0=0 \ \text{\(\lor\)} \ \exists \ m, \ 0=S \ m) \\
\mid \ S \ k \ \Rightarrow \ \text{inr} \ (\text{ex_intro} \ k \ \text{eq_refl}) \\
\quad : (S \ k = 0 \ \text{\(\lor\)} \ \exists \ m, \ S \ k = S \ m) \\
\text{end}
\]
Injectivity of constructors:

Definition pred (n:nat) :=
  match n with O => O | S k => k end.

f_equal pred : S n = S m -> n = m

Tactic injection H:
- applies this construction on hyp H: C t₁..tₙ = C u₁..uₙ
- derives proofs of t₁=u₁ .. tₙ=uₙ
Equational theory of \texttt{nat}

Discrimination of constructors:

\begin{verbatim}
Definition P (n:nat) :=
  match n return Prop with O => True | S k => False end.

match (e:0=1) in _=y return P y with
  | eq_refl => I : P 0 (* P 0 = True *)
end : P 1 (* P 1 = False *)
\end{verbatim}

Tactic \texttt{discriminate}:

- solves goals of the form \( C \ t_1 \ldots t_n \ <\!\!\!=\!\!\!= D \ u_1 \ldots u_k \)
- \texttt{discriminate \ H} solves the goal when \( H : C \ t_1 \ldots t_n = D \ u_1 \ldots u_k \)
Vectors (Lists with size)

Inductive type with parameters and index:

\[
\text{Inductive } \text{vect} (A:\text{Type}) : \text{nat} \rightarrow \text{Type} := \\
| \text{niln} : \text{vect} A \ O \\
| \text{consn} : \\
\quad A \rightarrow \forall n:\text{nat}, \text{vect} A \ n \rightarrow \text{vect} A \ (S \ n).
\]

which defines

- a family of types-predicates:
  \[ \Gamma \vdash \text{vect} : \text{Type} \rightarrow \text{nat} \rightarrow \text{Type} \]
- a set of introduction rules for the types in this family

\[
\Gamma \vdash A : \text{Type} \\
\overline{\Gamma \vdash \text{niln}_A : \text{vect} A \ O} \\
\Gamma \vdash A : \text{Type} \quad \Gamma \vdash a : A \quad \Gamma \vdash n : \text{nat} \quad \Gamma \vdash l : \text{vect} A \ n \\
\overline{\Gamma \vdash \text{consn}_A a \ n \ l : \text{vect} A \ (S \ n)}
\]
Vectors : elimination

- an elimination rule (pattern-matching operator with a result depending on the object which is eliminated)

\[
\begin{align*}
\Gamma \vdash v : \text{vect } A n & \quad \Gamma, p : \text{nat}, x : \text{vect } A p \vdash C(p, x) : s \\
\Gamma \vdash t_1 : C(O, \text{nil}_A) \\
\Gamma, a : A, n : \text{nat}, l : \text{vect } A n \vdash t_2 : C(S n, \text{cons}_A a n l)
\end{align*}
\]

\[
\Gamma \vdash \left( \text{match } v \text{ as } x \text{ in } \text{vect } \_ \_ p \text{ return } C(p, x) \text{ with } \begin{array}{l}
niln \Rightarrow t_1 \\
\text{cons} \ a \ n \ l \Rightarrow t_2
\end{array} \right) : C(n, v)
\]

- and the obvious reduction rules (\(\iota\)-reduction)
Well-formed inductive definitions
Constructors of the inductive definition \( I \) have type:

\[
\kappa : \forall (z_1 : C_1) \ldots (z_k : C_k). I \ a_1 \ldots a_n
\]

where \( C_i \) can feature instances of \( I \).
Question: can these instances be arbitrary?
Issues

Constructors of the inductive definition $I$ have type:

$$\kappa : \forall (z_1 : C_1) \ldots (z_k : C_k). I \ a_1 \ldots a_n$$

where $C_i$ can feature instances of $I$.

Question: can these instances be arbitrary? No!

Example:

```ocaml
Inductive lambda : Type :=
| Lam : (lambda -> lambda) -> lambda
```

```ocaml
Definition app (x y:lambda) := match x with (Lam f) => f y end.
Definition Delta := Lam (fun x => app x x).
Definition Omega := app Delta Delta.
```

and the evaluation of $\Omega$ loops.
Constructors of the inductive definition $I$ have type:

$$\kappa : \forall (z_1 : C_1) \ldots (z_k : C_k). I a_1 \ldots a_n$$

where $C_i$ can feature instances of $I$.

Question: can these instances be arbitrary? No!

Example:

```ocaml
Inductive lambda : Type :=
  | Lam : (lambda -> lambda) -> lambda

Definition app (x y:lambda) :=
  match x with (Lam f) => f y end.
Definition Delta := Lam (fun x => app x x).
Definition Omega := app Delta Delta.
```

and the evaluation of $\Omega$ loops.
Necessity of restrictions

Things can even be worse:

```plaintext
Inductive lambda : Type :=
| Lam : (lambda -> lambda) -> lambda
```

Now define:

```plaintext
Fixpoint lambda_to_nat (t : lambda) : nat :=
  match t with Lam f -> S (lambda_to_nat (f t)) end.
```
Necessity of restrictions

Things can even be worse:

\begin{verbatim}
Inductive lambda : Type :=
| Lam : (lambda -> lambda) -> lambda
\end{verbatim}

Now define:

\begin{verbatim}
Fixpoint lambda_to_nat (t : lambda) : nat :=
  match t with Lam f -> S (lambda_to_nat (f t)) end.
\end{verbatim}

What happens with \((\text{lambda_to_nat} (\text{Lam} (\text{fun} \ x \ => \ x)))\)?
The way out: (strict) positivity condition

- An inductive type is defined as the smallest type generated by a set \((\kappa_i)_{1 \leq i \leq n}\) of constructors.

- We can see it as \(\mu X, \oplus_{1 \leq i \leq n} \kappa_i(X)\) (with \(\mu\) a fixpoint operator on types).
  Eg: \(N = \mu X. 1 + X\) and so \(N = 1 + N\)

- The existence of this smallest type can be proved at the impredicative level when the operator \(\lambda X, \oplus_{1 \leq i \leq n} \kappa_i(X)\) is monotonic.
  \(\mu X : \mathbb{P}.(X \to A) \to A\) has a fixpoint...

- In order both to ensure monotonicity and to avoid paradox (predicativity of Type), Coq enforces a strict positivity condition: \(X\) should never appear on the left of an arrow in the type of its constructors.
The way out: (strict) positivity condition

More precisely, if the type (a.k.a arity) of a constructor is:

\[ c : C_1 \rightarrow \ldots \rightarrow C_k \rightarrow I \ a_1 \ldots \ a_k \]

it is well-formed when:

- \( I \ a_1 \ldots \ a_k \) is well-formed w.r.t. the uniformity of parametric arguments and typing constraints;
- \( I \) does not appear in any of the \( a_1, \ldots, a_k \);
- Each \( C_i \) should either not refer to \( I \) or be of the form:
  \[ C'_1 \rightarrow \ldots \rightarrow C'_m \rightarrow I \ b_1 \ldots \ b_k \]
  well typed and with no other occurrence of \( I \).

And the rule generalizes as such to dependent products (instead of arrow). Said otherwise:

\[ c : (\forall \Gamma_i, C_i) \rightarrow I \ a_1 \ldots \ a_k \text{ where } I \not\in FV(\Gamma_i) \text{ and } \]

\[ C_i = \begin{cases} I \ b_1 \ldots \ b_k & \text{where } I \not\in FV(b_1 \ldots b_k) \\ T & \text{otherwise} \end{cases} \]
There are more constraints, that will be explained later:

1. **predicativity/impredicativity**
   An inductive is predicative when all constructor argument types live in a sort not bigger than the declared sort for the inductive

2. **restriction on eliminations**
   when the predicativity condition is not satisfied
Girard’s paradox:

- **Type : Type**
- Generalizes to $X : \text{Type}$ with an embedding $\text{Type} \rightarrow X$

**Inductive** $e \ (A:s1) : s2 := C \ (_:A)$.

- $C : A \rightarrow e(A)$
- pattern-matching: $e(A) \rightarrow A$
- reduction: $C$ and pattern-matching are inverses

If $s_2 : s_1$, the paradox applies...

**Conclusion:** inductive definitions must be predicative, otherwise eliminations must be restricted (see Paulin’s Habilitation thesis)
Dependent pattern-matching

**Inductive** \( I \ (p:\text{Par}) : A \rightarrow s := \)
\[
\mid \kappa \ (x_1:C_1) \ldots (x_n:C_n) : I \ p \ u \\
\mid \ldots
\]

match \( t \) as \( h \) in \( I \ _ \ a \) return \( P(a,h) \) with
\[
\mid \kappa \ x_1 \ldots x_n \Rightarrow e \\
\ldots
\]
end

**Typing conditions:**
- \( \vdash t : I \ q \ b \)
- \( a : A[q/p], h : I \ q \ a \vdash P : s' \)
- \( x_1 : C_1[q/p], \ldots, x_n : C_n[q/p] \vdash e : P(u[q/p], \kappa \ q \ x_1 \ldots x_n) \)

Then the match has type \( P(b, t) \)
Tactics for case analysis

- `case t` is the most primitive. It:
  - generates a (proof) term of the form `match t with ...;`
  - guesses the return type from the goal (under the line);
  - does not introduce/name the arguments of the constructor by default, but there is a syntax for choosing names.

- The `case_eq` variant modifies the guessing of the return type so that equalities are generated.

- The `destruct` variant modifies the guessing of the return type so that it generalizes the hypotheses depending on `t`. The `destruct t eqn:H` variant allows to keep an equality `H` as well between `t` and each pattern.
The fixpoint operator (reduction)

Fixpoint expression with dependent result

\[(\text{fix } f (x : A) : B(x) := t(f, x))\]

- Typing

\[
\begin{align*}
& f : (\forall (x : A), B(x)), x : A \vdash t : B(x) \\
\implies & \vdash (\text{fix } f (x : A) : B(x) := t(f, x)) : \forall (x : A), B(x)
\end{align*}
\]
Fixpoint operator: well-foundness

Requirement of the Calculus of Inductive Constructions:
- the argument of the fixpoint has type an inductive definition
- recursive calls are on arguments which are structurally smaller

Example of recursor on natural numbers

\[
\begin{align*}
\lambda P : \text{nat} \rightarrow s, \\
\lambda H_O : P(O), \\
\lambda H_S : \forall m : \text{nat}, P(m) \rightarrow P(S m), \\
\text{fix } f (n : \text{nat}) : P(n) := \\
\quad \text{match } n \text{ as } y \text{ return } P(y) \text{ with} \\
\quad \quad O \Rightarrow H_O \mid S m \Rightarrow H_S m (f m) \\
\quad \text{end}
\end{align*}
\]

is correct with respect to CCI: recursive call on \( m \) which is structurally smaller than \( n \) in the inductive \( \text{nat} \).
The guard condition

Fixpoint operator : typing rules

\[ \Gamma \vdash l : s \quad \Gamma, x : l \vdash C : s' \quad \Gamma, x : l, f : (\forall x : l, C) \vdash t : C \quad t^0_f \triangleleft_l x \]

\[ \Gamma \vdash (\text{fix } f (x : l) : C := t) : \forall x : l, C \]

the main rules for \( t^\rho_f \triangleleft_l x \) are:

\[ z \in \rho \cup \{x\} \quad (u_i^\rho_f \triangleleft_l x)_{i=1}^n \quad A|_f^\rho \triangleleft_l x \quad (t_i^\rho_f \cup \{x \in \vec{x}_i | x : \forall y : U.1 \vec{u}\} \triangleleft_l x)_i \]

match \( z \ u_1 \ldots u_n \) return \( A \) with \( (c_i \vec{x}_i \Rightarrow t_i)_i \) end\( |_f^\rho \triangleleft_l x \)

\[ t \neq (z \vec{u}) \text{ for } z \in \rho \cup \{x\} \quad t^\rho_f \triangleleft_l x \quad A|_f^\rho \triangleleft_l x \quad (t_i^\rho_f \triangleleft_l x)_i \]

match \( t \) return \( A \) with \( (c_i \vec{x}_i \Rightarrow t_i)_i \) end\( |_f^\rho \triangleleft_l x \)

\[ y \in \rho \]

\[ f (y \ u_1 \ldots u_n)|_f^\rho \triangleleft_l x \]

\[ f \not\in \text{FV}(t) \quad t^\rho_f \triangleleft_l x \]

+ contextual rules ...
It covers simply the schema of primitive recursive definitions and proofs by induction which have recursive calls on all subterms.

\[
\begin{align*}
\lambda P &: \text{list } A \to s, \\
\lambda f_1 &: P \text{ nil,} \\
\lambda f_2 &: \forall (a : A)(l : \text{list } A), P l \to P (\text{cons } a l), \\
\text{fix } Rec (x : \text{list } A) &: P x := \\
&\quad \text{match } x \text{ return } P x \text{ with} \\
&\quad \quad \quad \text{nil } \Rightarrow f_1 | (\text{cons } a l) \Rightarrow f_2 a l (Rec l) \\
\end{align*}
\]

has type

\[
\begin{align*}
\forall P &: \text{list } A \to s, \\
P \text{ nil,} \to \\
(\forall (a : A)(l : \text{list } A), P l \to P (\text{cons } a l)) \to \\
\forall (x : \text{list } A), P x
\end{align*}
\]
Remarks on the criteria

Possibility of recursive call on deep subterms

```coq
Fixpoint mod2 (n:nat) : nat :=
  match n with
    O => O | S O => S O
    | S (S x) => mod2 x
  end
```

Possibility of recursive call on terms build by case analysis if each branch is a strict subterm (actual rule very complex!):

```coq
Definition pred (n:nat) : n<>0->nat:=
  match n return n<>0->nat with
    S p => (fun (h:S p<>0) => p)
    | O => (fun (h:0<>0) =>
              match h (refl_equal 0) return nat with end)
  end

Fixpoint F (n:nat) : C :=
  match iszero n with
    (left H (*H:n=O*)) => ...
    | (right H (*H:n<>0*)) => F (pred n H)
  end
```
Remarks on the criteria

Note: only the recursive arguments with the same type are considered recursive (otherwise paradox related to impredicativity)

\[
\begin{align*}
\text{Definition } & \ ID : \ Prop := \forall (A:Prop), \ A \rightarrow A. \\
\text{Definition } & \ id : \ ID := \text{fun } A \ x \Rightarrow x. \\
\text{Inductive } & \ Singl : \ Prop := c (_:ID). \ (* \ non \ recursive \ *) \\
\text{Fixpoint } & \ f (x : Singl) : \ bool := \\
& \quad \text{match } x \ \text{with} \ (c \ a) \Rightarrow f (a \ Singl \ x) \ \text{end.} \\
& f (c \ id) \xrightarrow{\lambda} f (id \ Singl (c \ id)) \xrightarrow{\beta} f (c \ id)
\end{align*}
\]
Tactics for induction

fix f <n>, where <n> is a numeral is the most primitive. It:

- generates a (proof) term of the form:
  fun g1 g2 => fix f h1 h2 t h3 {struct t} := ?F h1 h2 t

where:

- g1, g2 are the objects in the context (above the line);
- h1, h2, t, h3 are the objects quantified in the goal (under the line);
- ?F can call f (= recursive calls);
- the termination of f is should eventually be guaranteed by structural recursion on t;

Qed checks the well-formedness, which was not guaranteed so far: error messages come late and may be difficult to interpret.
Tactics for induction

`elim t` applies an induction scheme, i.e. a lemma of the form:

\[ \forall P : T \rightarrow \text{Type}, \ldots \rightarrow \forall t' : T, P t' \]

- It guesses argument \( P \) from the goal (under the line), abstracting all the occurrences of \( t \).
- It guesses the elimination scheme to be used (\( T_{\text{ind}}, T_{\text{rect}}, \ldots \)) from the sort of the goal and the type of \( t \).
- The `elim t using S` variant allows to provide a custom elimination scheme (or lemma!) \( S \), with the same unification heuristic.

- The `induction t` tactic guesses argument \( P \) taking into account the possible hypotheses depending on \( t \) present in the context (above the line). Plus it can introduce and name things automatically.

Remark: the `rewrite` tactic does a similar guessing job...
Fixpoint expansion

We would expect the usual expansion rule for fixpoints:

\[
\text{(fix } f \ (x:A): B \ x := t(f,x)\text{)e} \\
\rightarrow t(\text{fix } f \ (x:A): B \ x := t(f,x), \ e)
\]
We would expect the usual expansion rule for fixpoints:

\[(\text{fix } f \ (x: A): B \ x := t(f, x)) e\]

\[\rightarrow t(\text{fix } f \ (x: A): B \ x := t(f, x), e)\]

... but this leads to infinite unfolding (SN broken)
Fixpoint expansion

We would expect the usual expansion rule for fixpoints:

\[(\text{fix } f (x:A): B x := t(f,x))e \]
\[\rightarrow t(\text{fix } f (x:A): B x := t(f,x), e)\]

... but this leads to infinite unfolding (SN broken)

Solution: allow this reduction only when \(e\) is a constructor

Beware:

- Guard condition ensures consistency (meaningful definition)
- Expansion restriction imposes a strategy
Advanced features of inductive types

- Prop vs Type
- Impredicative inductive definitions