MPRI 2-7-2: Proof Assistants

Bruno Barras, Matthieu Sozeau

Jan 12, 2017
Recap: Inductive Types and Elimination Rules

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):

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 =
dependent pattern-matching + guarded fixpoint
Logical connectives

Logical connectives and their non-dependent elimination schemes:

**Inductive** True : Prop := I.
  True_rect : forall P:Type, P -> True -> P.

**Inductive** False : Prop := .
  False_rect : forall P:Type, False -> P

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

**Inductive** or (A B:Prop) : Prop :=
  or_introl (=:A) | or_intror (=:B).
  or_ind : forall (A B P:Prop), (A->P) -> (B->P) -> P.
Plan

Inductive families
  Predicate defined by inference rules
  Definition of equality
  Vectors

Non-uniform parameters

Theory of Inductive types
  Strict Positivity
  Dependent pattern-matching
  Guarded fixpoint
  The guardedness check
Limitations of parameters

Defining a predicate:

\[
\text{Inductive even (n:nat) : Prop := even}_i \text{ (half:nat) (_, half+half=n).}
\]

Inductive types with parameters are some kind of “template”

\[
\text{Inductive listnat := nilnat | consnat (_, nat) (_, listnat).}
\]

\[
\text{Inductive listbool := nilbool | consbool (_, bool) (_, listbool).}
\]

No dependency between both types.

But in the definition of \text{even:nat->Prop} as an inductive type/set

\[
\text{even (S (S O)) depends on even 0.}
\]
Inductive families

Family = indexed type
\( P : \text{nat} \rightarrow \text{Type} \) represents the type family \((P(n))_{n \in \mathbb{N}}\)

Inductive family:

- Constructors do not inhabit uniformly the members of the family
- Recursive arguments can change the value of the index

Even numbers:

\[
\text{Inductive even : nat} \rightarrow \text{Prop} :=
\begin{align*}
\text{E0} : \text{even 0} \\
| \text{ESS (n:nat) (e:even n)} : \text{even (S (S n))}.
\end{align*}
\]

Syntax very close to inference rules!
Elimination scheme: minimality of predicate, rule-induction

`even_ind : forall (P:nat->Prop),
P O -> (forall n, P n -> P (S (S n))) ->
forall n, even n -> P n.`

Seems the analogous of nat’s dependent scheme
Elimination scheme

Elimination scheme: minimality of predicate, rule-induction

even_ind : forall (P:nat->Prop),
P 0 -> (forall n, P n -> P (S (S n))) ->
forall n, even n -> P n.

Seems the analogous of nat’s dependent scheme

Even’s dependent scheme (refers to constructors E0 and ESS):

forall (P : forall n, even n -> Prop),
P 0 E0 ->
(forall n (e:even n), P n e -> P (S (S n)) (ESS n e)) ->
forall n (e:even n), P n e

Definable in Coq, but not automatically generated (why? wait and see...)
Defining the dependent elimination scheme

Even more complex return clause: in

\[
\text{Definition even\textunderscore ind\textunderscore dep} \quad \text{(P : } \forall n \rightarrow \text{even } n \rightarrow \text{Prop)} \\
(h0 : P \ 0 \ \text{E0}) \\
(hSS : \forall n \ e, P \ n \ e \rightarrow P \ (S \ (S \ n)) \ (\text{ESS} \ n \ e)) \\
: \forall \ n, \ \text{even } n \rightarrow P \ n := \\
\text{fix } F \ n \ e := \\
\text{match } e \ \text{as } e' \ \text{in even } k \ \text{return } P \ k \ e' \ \text{with} \\
| \ \text{E0} \Rightarrow h0 : \ P \ 0 \ \text{E0} \\
| \ \text{ESS } k \ e' \Rightarrow \\
\quad hSS \ k \ e' \ (F \ k \ e') : \ P \ (S \ (S \ k)) \ (\text{ESS} \ k \ e') \\
\text{end}
\]

\text{Notation as } e' \ \text{in even } k \ \text{return } P \ k \ e' \ \text{is just a way to write the term } \text{fun } k \ e' \Rightarrow P \ k \ e' . \\
\text{Becomes natural with time...}
Equality: the paradigmatic indexed family

Propositional equality is defined as:

\[
\text{Inductive} \ eq \ (A : \text{Type}) \ (a : A) : A \to \text{Prop} := \\
\quad \text{eq}_\text{refl} : eq A a a.
\]

Notation "\(x = y\)" := (eq x y).

Its dependent elimination principle is of the form:

\[
\Gamma \vdash e : eq A t u \quad \Gamma, y : A, e' : eq A t y \vdash C(y, e') : s \\
\Gamma \vdash t : C(t, \text{eq}_\text{refl}_{A,t})
\]

\[
\frac{\text{match } e \text{ as } e' \text{ in } eq_\_y \text{ return } C(y, e') \text{ with } \text{eq}_\text{refl} \Rightarrow t}{\Gamma \vdash \text{end} : C(u, e)}
\]
Tactics related to equality

Tactics:

- **f_equal** (congruence) \[
  \begin{align*}
  x &= y \\
  f(x) &= f(y)
  \end{align*}
\]

- **discriminate** (constructor discrimination) \[
  \begin{align*}
  C(t_1, \ldots, t_n) &= D(u_1, \ldots, u_k) \\ \\
  \text{A}
  \end{align*}
\]

- **injection** (injectivity of constructors) \[
  \begin{align*}
  C(t_1, \ldots, t_n) &= C(u_1, \ldots, u_n) \\ \\
  t_1 &= u_1 & \ldots & t_n &= u_n
  \end{align*}
\]

- **inversion** (necessary conditions) \[
  \text{even } (S(Sn)) \\
  \text{even n}
\]

- **rewrite** (substitution) \[
  \begin{align*}
  x &= y \\
  P(y) &= P(x)
  \end{align*}
\]

- **symmetry, transitivity**
Inductive types with parameters and index

Example of vectors with size

Inductive vect (A:Type) : nat -> Type :=
| niln : vect A O
| consn :
    A -> forall n:nat, vect A n -> 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

\[
\begin{align*}
\Gamma & \vdash A : \text{Type} \\
\Gamma & \vdash \text{niln}_A : \text{vect } A \ O \\
\Gamma & \vdash \text{consn}_A a n l : \text{list } A \ (S \ n)
\end{align*}
\]
Inductive types with parameters and index

vectors : elimination

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

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

\[
\Gamma \vdash \left(\begin{array}{c}
\text{match } v \text{ as } x \text{ in } vect \_ \ p \text{ return } C(p, x) \text{ with }\\
\text{niln } \Rightarrow t_1 \ | \ \text{consn } a \ n \ l \Rightarrow t_2 \\
\text{end} \\
\end{array}\right) : C(n, v)
\]
Inhuctive types with parameters and index

- reduction rules preserve typing ($\iota$-reduction)

\[
\begin{align*}
\text{match } \text{niln}_A \text{ as } x \text{ in } \text{vect} \_p \text{ return } C(x, p) \text{ with } \\
\text{niln } \Rightarrow t_1 | \text{consn } a n l \Rightarrow t_2 \\
\text{end} \\
\rightarrow_\iota t_1
\end{align*}
\]

\[
\begin{align*}
\text{match } \text{consn}_A a' n' l' \text{ as } x \text{ in } \text{vect} \_p \text{ return } C(x, p) \text{ with } \\
\text{niln } \Rightarrow t_1 | \text{consn } a n l \Rightarrow t_2 \\
\text{end} \\
\rightarrow_\iota t_2[a', n', l' / a, n, l]
\end{align*}
\]
Non-uniform parameters

Non-uniform parameter:
- Like parameters: uniform conclusion
- Like indices: value can change in recursive subterms

\[\text{Inductive}\ \text{tuple} \ (A:\text{Type}) :=\]
\[\mid \text{H0} \ (_:A)\]
\[\mid \text{HS} \ (_:\text{tuple} \ (A\times A)).\]

\[\text{Definition}\ \text{t4} : \text{tuple} \ \text{nat} :=\]
\[\text{HS} \ \text{nat} \ (\text{HS} \ (\text{nat}\times\text{nat}) \ (\text{H0} \ _ \ ((1,2),(3,4)))).\]
Elimination rules

Pattern-matching:

\[
\begin{align*}
\Gamma & \vdash e : \text{tuple } A & \Gamma, h : \text{tuple } A & \vdash P(h) : s \\
\Gamma, x : A & \vdash t_0 : P(H_0 A x) & \Gamma, h : \text{tuple}(A \times A) & \vdash t_S : P(HS A h)
\end{align*}
\]

\[
\Gamma \vdash \begin{cases}
\text{match } e \text{ as } h \text{ return } P(h) \text{ with} \\
H_0 x \Rightarrow t_0 \\
HS h \Rightarrow t_S \\
\text{end}
\end{cases}
: P(e)
\]

Elimination:

tuple_rect :

\[
\forall (P : \forall A, \text{tuple } A \to \text{Type}), \\
(\forall A x, P A (H_0 A x)) \to \\
(\forall A h, P (A \times A) h \to P A (HS A h)) \to \\
\forall A (h : \text{tuple } A), P A h.
\]

Non-uniform parameters:

- In pattern-matching, behaves like a parameter
- In recursive principles, behaves like an index
Encoding inductive families

Non-uniform parameters can encode inductive families:

```plaintext
Inductive even (n:nat) : Prop :=
  E0' (_:n=0)
| ESS' (k:nat) (e:even k) (_:n=S (S k)).
Definition E0 : even 0 := E0' 0 eq_refl.
Definition ESS n e : even (S (S n)) :=
  ESS' (S (S n)) n e eq_refl.
```
Well-formed inductive definitions
Issues

Constructors of the inductive definition $I$ have type:

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

where $C_i$ can feature instances of $I$.
Question: can these instances be arbitrary?
Issues

Constructors of the inductive definition $I$ have type:

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

where $C_i$ can feature instances of $I$.
Question: can these instances be arbitrary?
Example:

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

Issues

Constructors of the inductive definition $I$ have type:

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

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

Question: can these instances be arbitrary?

Example:

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

We define:

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.
\end{verbatim}

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

Things can even be worse:

\textbf{Inductive lambda : Type :=}
| Lam : (lambda -> lambda) -> lambda

\textbf{Now define:}

\textbf{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:

\[
\text{Inductive lambda : Type :=} \\
| \text{Lam : (lambda -> lambda) -> lambda} \\
\]

Now define:

\[
\text{Fixpoint lambda_to_nat (t : lambda) : nat :=} \\
| \text{match t with Lam f -> S (lambda_to_nat (f t)) end.} \\
\]

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

- An inductive type is defined as the smallest type generated by a set \((\Gamma_i)_{1 \leq i \leq n}\) of constructors.
- We can see it as \(\mu X, \bigoplus_{1 \leq i \leq n} \Gamma_i(X)\) (with \(\mu\) a fixpoint operator on types).
- The existence of this smallest type can be proved at the impredicative level when the operator \(\lambda X, \bigoplus_{1 \leq i \leq n} \Gamma_i(X)\) is monotonic.
- In order both to ensure monotonicity and to avoid paradox, 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:

\[ c : C_1 \to \ldots \to C_k \to I_{\alpha_1 \ldots \alpha_k} \]

it is well-formed when:

- \( I_{\alpha_1 \ldots \alpha_k} \) is well-formed w.r.t. the uniformity of parametric arguments and typing constraints;
- \( I \) does not appear in any of the \( \alpha_1, \ldots, \alpha_k \);
- Each \( C_i \) should either not refer to \( I \) or be of the form:

\[ C'_1 \to \ldots C'_m \to I_{b_1 \ldots b_k} \]

well typed and with no other occurrence of \( I \).

And the rule generalizes as such to dependent products (instead of arrow).
More well-formation conditions...

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
Dependent pattern-matching

\[
\text{Inductive } I \ (p : \text{Par}) : A \rightarrow s := \\
\quad \ldots \ | \ \Gamma (x_1 : C_1) \ldots (x_n : C_n) : I \ p \ u \\
\quad \ldots
\]

match \( t \) as \( h \) in \( I \ \_ \ a \) return \( P(a,h) \) with

\[
\ldots \\
\quad \Gamma \ x_1 \ldots x_n => e \\
\ldots \end{match}

Typing conditions:

\[\vdash t : I \ q \ a\]

\[a : A[q/p], h : I \ q \ a \vdash P : s'\]

\[x_1 : C_1[q/p], \ldots, x_n : C_n[q/p] \vdash e : P(u[q/p], \Gamma q \ x_1 \ldots x_n)\]

Then the match has type \( P(a, t) \)
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 fixpoint operator (reduction)

Fixpoint expression with dependent result

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

Typing

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

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

is correct with respect to CCI: recursive call on \( m \) which is structurally smaller than \( n \) in the inductive \text{nat}.
Fixpoint operator : typing rules

\[
\Gamma \vdash \ell : s \quad \Gamma, x : A \vdash C : s' \quad \Gamma, x : \ell, f : (\forall x : \ell, C) \vdash t : C \quad t_\rho^f \not< x
\]
\[
\Gamma \vdash (\text{fix } f (x : \ell) : C ::= t) : \forall x : \ell, C
\]

the main definition of \( t_\rho^f < x \) are:

\[
z \in \rho \cup \{x\} \quad (u_i^\rho < x)_{i=1\ldots n} \quad A^\rho < x \quad (t_i^\rho \{x \in \bar{x} | x : \forall y : U.1 \bar{u}\} < x)_{i}
\]

match \( z \ u_1 \ldots u_n \) return \( A \) with \( \ldots c_i \bar{x}_i \Rightarrow t_i \ldots \) end \( \rho^f < x \)

\[
t \neq (z \bar{u}) \text{ for } z \in \rho \cup \{x\} \quad t^\rho < x \quad A^\rho < x \quad \ldots t_i^\rho < x \ldots
\]

match \( t \) return \( A \) with \( \ldots c_i \bar{x}_i \Rightarrow t_i \ldots \) end \( \rho^f < x \)

\[
y \in \rho \quad \frac{f \ y^\rho < x}{f \ y^\rho < x} \quad \frac{f \notin t}{t^\rho < x}
\]

+ contextual rules \ldots
Remarks on the criteria

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

\[
\lambda P : \text{list } A \rightarrow s, \\
\lambda f_1 : P \text{ nil}, \\
\lambda f_2 : \forall(a : A)(l : \text{list } A), P l \rightarrow P(\text{cons } a l),
\]

fix \( \text{Rec} (x : \text{list } A) : P x \) :=

match \( x \) return \( P x \) with

\( \text{nil} \) \( \Rightarrow f_1 \mid (\text{cons } a l) \Rightarrow f_2 a l(\text{Rec } l) \)

end

- has type

\[
\forall P : \text{list } A \rightarrow s, \\
P \text{ nil}, \rightarrow \\
(\forall(a : A)(l : \text{list } A), P l \rightarrow P(\text{cons } a l)) \rightarrow \\
\forall(x : \text{list } A), P x
\]
Remarks on the criteria

Possibility of recursive call on deep subterms

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:

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:n=O)) => ...
| (right (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)

Inductive Singl (A:Prop) : Prop := c : A -> Singl A.
Definition ID : Prop := forall (A:Prop), A -> A.
Definition id : ID := fun A x => x.
Fixpoint f (x : Singl ID) : bool :=
    match x with (c a) => f (a (Singl ID) (c ID id)) end.

\[ f(c \text{ ID id}) \rightarrow f(id(\text{Singl ID})(c \text{ ID id})) \rightarrow f(c \text{ ID id}) \]
Tactics for induction

`fix <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 -> Type, .... -> 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 (\( T_{ind}, \) \( T_{rect} \),...) 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:

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