MPRI 2-7-2: Proof Assistants

Bruno Barras

Oct 4, 2019
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):

\[ \text{nat_rect} : \forall P : \text{nat} \rightarrow \text{Type}, \]
\[ P \ O \rightarrow (\forall n, P \ n \rightarrow P \ (S \ n)) \rightarrow \]
\[ \forall n, P \ n. \]
\[ := \text{fun} P \ h0 \ hS \Rightarrow \text{fix} F \ n := \]
\[ \text{match} n \ \text{return} \ P \ n \ \text{with} \]
\[ | \ O \Rightarrow h0 \]
\[ | \ S \ k \Rightarrow hS \ k \ (F \ k) \]
\[ \text{end} \]

Eliminator of recursive type =
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 -> 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.
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

\textbf{Inductive} \quad \texttt{eq \ (A : Type) \ (x : A) : A \to Prop :=}
\begin{align*}
| \ \texttt{eq_refl : eq A x x}.
\end{align*}

\textbf{Elimination:}

- \texttt{eq_rect:} \quad \forall \ A \ x \ (P : A \to Type), \ P \ x \to \forall \ y, \ x = y \to P \ y

- \textbf{Dependent version (generated by \texttt{Scheme}):}

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

```
match n return n=0 \ / \ \exists m, n=S m with
| 0 => inl eq_refl : (0=0 \ / \ \exists m, 0=S m)
| S k => inr (ex_intro k eq_refl)
   : (S k = O \ / \ \exists m, S k = S m)
end
```
Injectivity of constructors:

\[\text{Definition} \quad \text{pred} \ (n:\text{nat}) := \]
\[\quad \text{match} \ n \ \text{with} \ O \Rightarrow O \ |
\quad S \ k \Rightarrow k \ \text{end}.\]

\[\text{f_equal \ pred} : \ S \ n = S \ m \rightarrow n = m\]

**Tactic** \textit{injection H}:

- applies this construction on hyp \(H: C \ t_1 \ldots t_n = C \ u_1 \ldots u_n\)
- derives proofs of \(t_1 = u_1 \ldots t_n = u_n\)
Equational theory of \texttt{nat}

**Discrimination of constructors:**

\[
\text{Definition } P \ (n:\texttt{nat}) := \\
\quad \text{match } n \ \text{return Prop with } \begin{array}{ll}
O & \Rightarrow \text{True} \\
S \ k & \Rightarrow \text{False}
\end{array} \end{equation}
\]

\[
\text{match } (e:0=1) \ \text{in } _=y \ \text{return } P \ y \ \text{with} \\
\quad \text{| eq_refl } \Rightarrow I : P \ 0 \ (\ast P \ 0 = \text{True } \ast) \\
\text{end } : P \ 1 \ (\ast P \ 1 = \text{False } \ast)
\]

**Tactic \texttt{discriminate}:**

- solves goals of the form \( C \ t_1 \ldots t_n \ \text{<>} \ D \ u_1 \ldots u_k \)
- \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:

\[
\text{Inductive } \text{vect } (A:\text{Type}) : \text{nat} \rightarrow \text{Type} := \\
\mid \text{niln} : \text{vect } A \ 0 \\
\mid \text{consn} : A \rightarrow \forall n:\text{nat}, \text{vect } A\ n \rightarrow \text{vect } A\ (S\ n).
\]

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

\[
\frac{\Gamma \vdash A : \text{Type}}{\Gamma \vdash \text{niln}_A : \text{vect } A \ 0}
\]

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

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

\[
\begin{align*}
\Gamma &\vdash v : vect\ A\ n & \Gamma, p : nat, x : vect\ A\ p &\vdash C(p, x) : s \\
\Gamma &\vdash t_1 : C(O, \text{nil}_A) \\
\Gamma, a : A, n : nat, l : vect\ A\ n &\vdash t_2 : C(S\ n, \text{cons}_A\ a\ n\ l) \\
\Gamma &\vdash \left(\begin{array}{l}
\text{match}\ v\ \text{as}\ x\ \text{in}\ vect\ _\ p\ \text{return}\ C(p, x)\ \text{with}\\
\text{niln} \Rightarrow t_1 | \text{consn}\ a\ n\ l \Rightarrow t_2 \\
\text{end}\end{array}\right) : C(n, v)
\end{align*}
\]

- and the obvious reduction rules ($\iota$-reduction)
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?
Constructors of the inductive definition $\text{I}$ have type:

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

where $C_i$ can feature intances of $\text{I}$.

Question: can these instances be arbitrary? No!

Example:

```plaintext
Inductive lambda : Type :=
| Lam : (lambda -> lambda) -> lambda
```
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? No!

Example:

```plaintext
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:

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

Now define:

\[
\text{Fixpoint } \lambda_{\rightarrow} \text{nat : } (t : \lambda) : \text{nat := } \\
\quad \text{match } t \text{ with Lam } f \rightarrow S (\lambda_{\rightarrow} \text{nat } (f t)) \text{ end.}
\]
Necessity of restrictions

Things can even be worse:

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

Now define:

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

What happens with `(lambda_to_nat (Lam (fun x => x)))`?
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).

Eg: $\mathbb{N} = \mu X. 1 + X$ and so $\mathbb{N} = 1 + \mathbb{N}$

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.

$\mu X : \mathbb{P}.(X \rightarrow A) \rightarrow A$ has a fixpoint...

In order both to ensure monotonicity and to avoid paradox (predicativity of $\text{Type}$), 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:
\[
c : C_1 \to \ldots \to C_k \to I \ a_1 \ldots a_k
\]
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:
  \[
  C'_1 \to \ldots \ C'_m \to I \ b_1 \ldots b_k
  \]
  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**
   When the predicativity condition is not satisfied
Size paradoxes

Girard’s paradox:

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

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

- $C : A \to e(A)$
- pattern-matching: $e(A) \to A$
- 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)
Inductive $I (p : \text{Par}) : A \rightarrow s :=$
| $\kappa (x_1 : C_1) \ldots (x_n : C_n) : I \ p \ u$
| $\ldots$

match $t$ as $h$ in $I \ _ \ a$ return $P(a, h)$ with
| $\kappa \ x_1 \ldots \ x_n \Rightarrow e$
| $\ldots$
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 fixpoint operator (reduction)

Fixpoint expression with dependent result

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

- Typing

\[ f : (\forall (x : A), B(x)), x : A \vdash t : B(x) \]

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

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

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

\[ \Gamma \vdash I : s \quad \Gamma, x : A \vdash C : s' \quad \Gamma, x : I, f : (\forall x : I, C) \vdash t : C \quad t^{\emptyset}_f <_I x \]

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

the main definition of \( t^{\rho}_f <_I x \) are:

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

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

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

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

\[ y \in \rho \quad f \neq \text{FV}(t) \]

\[ f (y \ u_1 \ldots u_n)^{\rho}_f <_I x \quad t^{\rho}_f <_I x \]

+ contextual rules ...
Remarks on the criteria

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

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

- has type

\[
\begin{align*}
\forall P : \text{list } A &\rightarrow s, \\
P \text{ nil}, &\rightarrow \\
(\forall (a : A)(l : \text{list } A), P l &\rightarrow P (\text{cons } a l)) \rightarrow \\
\forall (x : \text{list } A), P x
\end{align*}
\]
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{Inductive} & \quad \text{Singl} \ (A: \text{Prop}) : \text{Prop} := c : A \rightarrow \text{Singl} \ A. \\
\text{Definition} & \quad \text{ID} : \text{Prop} := \text{forall} \ (A: \text{Prop}), A \rightarrow A. \\
\text{Definition} & \quad \text{id} : \text{ID} := \text{fun} \ A \ x \Rightarrow x. \\
\text{Fixpoint} & \quad f \ (x : \text{Singl} \ \text{ID}) : \text{bool} := \\
& \quad \quad \text{match} \ x \ \text{with} \ (c \ a) \Rightarrow f \ (a \ (\text{Singl} \ \text{ID}) \ (c \ \text{ID} \ \text{id})) \ \text{end}. \\
\end{align*}

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

\texttt{fix} <n>, where <n> is a numeral is the most primitive. It:

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

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 \(t\);

\text{Qed} checks the well-formedness, which was not guaranteed so far: error messages come late and may be difficult to interpret.
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 ($T_{\text{ind}}$, $T_{\text{rect}}$, ...) 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...
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

Beware:

- Guard condition ensures consistency (meaningful definition)
- Expansion restriction imposes a strategy
Next week...

Advanced features of inductive types

- Prop vs Type
- Impredicative inductive definitions